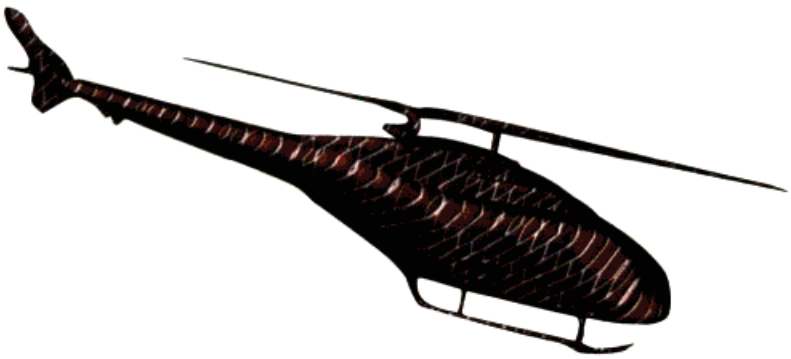
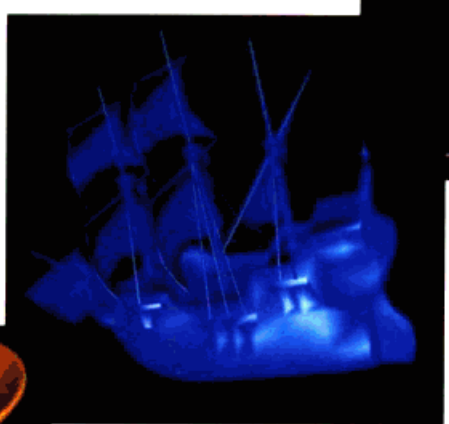
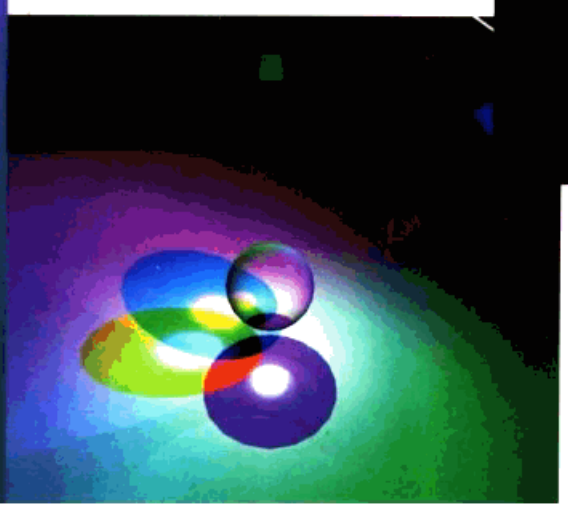


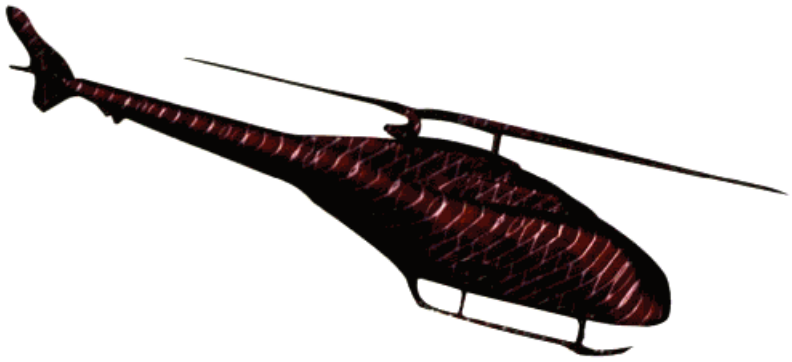
OpenGL编程 入门与提高

贾志刚／编著

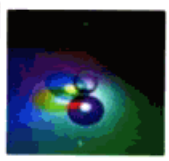


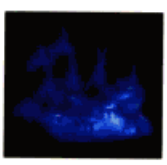


中国环境科学出版社



OpenGL 编程 入门与提高





贾志刚／编著

ISBN 7-80135-804-X

9 787801 358042 >

ISBN 7-80135-804-X/Z · 379

定价：42.00元（含磁盘）

前　言

OpenGL 是由 SGI 公司为其工作站开发的一个工业标准计算机图形软件接口，并由 Microsoft 集成到 Windows NT 和 Windows 95 中，适用于多种硬件平台及操作系统。用户可以方便地利用这个图形库创建高质量的三维彩色图像及三维动画。OpenGL 早期用于科研上的建模、模拟等方面，随着个人计算机性能的迅速提高，已经广泛地用于 CAD、可视化、计算机游戏等方面的三维图形制作、动画设计。OpenGL 作为一个热门的三维图形软件已经得到了程序设计人员的认可。

作为国内第一本详细介绍 OpenGL 资料，《精通 OpenGL》（贾志刚编著，电子工业出版社 1998 年 8 月出版）讲述了 OpenGL 的程序设计及函数的使用方法，但是，该书没有提供详细的计算机图形学的背景知识，也没有详细地介绍在 Windows 环境下的程序设计，给初学 OpenGL 的读者带来了学习、认识上的不方便。本书就是为解决上述问题编写的。

本书注重对计算机图形学知识及 OpenGL 编程技巧的介绍，结合大量图例对一些复杂的计算机图形学概念做深入浅出的解释，使初学者能够全面、深入地理解本书的内容，并在具体的编程过程中达到一定的水平。另外，本书还详细介绍如何在 Windows 环境下使用 Visual C++ 进行 OpenGL 程序设计，同时提供使用 Visual Basic, Delphi 语言进行 OpenGL 编程的具体指导。本书是一本比较理想的 OpenGL 入门书，对于要把 OpenGL 应用程序从工作站移植到 Microsoft Windows 环境的程序员也有帮助。为了便于读者使用本书的程序，本书提供包含所有源程序及相应的可执行文件软盘。这些程序均经过严格的检查、调试。

本书的内容可以分为如下部分：

第一部分作为学习、使用 OpenGL 的预备知识。介绍计算机图形学及 OpenGL 的基本概念。OpenGL 的工作原理、三维计算机图形学基础、三维坐标系统及在计算机屏幕上的实现过程。

第二部分是学习 OpenGL API 的基础。结合一些简单的程序，介绍借助 OpenGL 的 AUX 库在程序中调用 OpenGL 函数。结合具体程序，详细介绍、分析在 Windows 环境下调用 OpenGL 的过程，以及如何查找错误、调试 OpenGL 程序。

第三部分为 OpenGL 的核心内容。介绍使用 OpenGL 绘制具有色彩及光照的三维物体的基本内容，这些内容包括：OpenGL 几何要素的使用、设置物体的颜色、OpenGL 的矩阵变换、设置光源及材质的属性、绘制有光照的场景。作为使用 OpenGL 更深入的内容，介绍绘制有特殊视觉效果的图像，这些特殊效果包括：获得半透明效果的混合操作、绘制反走样图形、在场景中使用雾效果、进行纹理映射，在物体上绘制纹理、对物体的抖动操作、利用累加缓冲区产生的运动模糊、得到景深效果等。绘制 NURBS 曲线、曲面；为了设计交互式的 OpenGL 程序，将介绍 OpenGL 的选择、反馈机制。

第四部分内容详细介绍如何实现基于 MFC 的 OpenGL 编程，以及在编程过程中的一些注意事项。介绍如何使用 OCX 控件，实现在 Microsoft Visual Basic 和 Borland Dephi 语言环

境下，调用 OpenGL 函数。

参加本书编写工作的同志还有：王群、赵双勤、张砚臣、左键、田志伟、朱文羽、沈金来、郑文宇，在此表示感谢。

在本书的编写、校对以及出版过程中，得到了环境科学出版社的大力支持与帮助，在此表示诚挚的谢意。

由于编写时间紧迫，书中疏漏与错误难免，恳请用户和读者予以指正。反馈意见请发至电子信箱：jiazg@cnscpv.org.cn

作 者

1999 年 4 月

目　录

第一章　OpenGL 简介……1
1.1　关于 OpenGL……1
1.2　OpenGL 的历史……1
1.3　OpenGL 的工作原理……2
1.3.1　图形体系——软件与硬件……2
1.3.2　基本实现的限制……3

第二章　三维图形学基础……4
2.1　三维感受……4
2.2　坐标系统……6
2.3　投影三维的实质……9
2.4　本章小结……9

第三章　使用 AUX 库函数……10
3.1　OpenGL API……10
3.1.1　OpenGL API 的构成……10
3.1.2　OpenGL 数据类型……11
3.1.3　函数命名规则……11
3.2　AUX 库函数……12
3.3　解剖一个 OpenGL 小程序……12
3.4　用 OpenGL 绘制几何形状……16
3.5　用 AUX 做动画……22
3.6　相关函数……27

第四章　Windows 环境下的 OpenGL……29
4.1　在 Windows 窗口下绘图……29
4.1.1　GDI 设备描述表……29
4.1.2　OpenGL 绘图描述表……31
4.2　为 OpenGL 准备窗口……34
4.3　Windows 环境下的 OpenGL 程序……36
4.4　OpenGL 程序的调试……42
4.5　OpenGL 的扩展……44
4.6　相关函数……44

第五章　在三维空间绘制几何要素……46
5.1　在三维空间画点……46
5.2　在三维空间画线……52
5.3　在三维空间画三角形……58
5.4　创建三维实体……60
5.5　其他几何要素……67
5.6　相关函数……71

第六章　坐标变换……73
6.1　对坐标变换的理解……73
6.2　OpenGL 的矩阵操作……75
6.3　矩阵操作……77
6.4　使用投影……81
6.5　一个例子……84
6.6　相关函数……87

第七章　颜色和明暗处理……88
7.1　计算机显示的颜色……88
7.2　选择颜色……90
7.3　Windows 环境下的调色板……92
7.4　创建调色板……94
7.5　色彩指数模式……98
7.6　相关函数……101

第八章　光照处理……102
8.1　光源种类……102
8.2　材质属性……103
8.3　在场景中加入光照……105
8.4　阴影处理……121
8.5　光照处理与色彩指数模式……127
8.6　相关函数……129

第九章　在 OpenGL 中使用光栅图形……130
9.1　绘制位图……130
9.2　Pixmaps——彩色位图……134
9.3　位图浏览器……140
9.4　相关函数……147

第十章　纹理映射……149
10.1　纹理映射基础……149

10.2 绘制纹理化的多边形……153
10.3 多重纹理……154
10.4 自动生成纹理坐标……155
10.5 相关函数……159

第十一章 绘制三维物体……160
11.1 创建一个二次曲面……160
11.2 Bezier 曲线和曲面……162
11.3 NURBS……172
11.4 多边形镶嵌……177
11.5 相关函数……183

第十二章 使用缓冲区……185
12.1 什么是缓冲区?……185
12.2 颜色缓冲区……188
12.3 深度缓冲区……189
12.4 模板缓冲区……199
12.5 累加缓冲区……204
12.6 相关函数……209

第十三章 视觉效果:混合、雾……210
13.1 混 合……210
13.1.1 半透明效果……210
13.1.2 反走样操作……215
13.1.3 一个画笔程序……215
13.2 雾效果……227
13.2.1 绘制深度暗示的茶壶……227
13.2.2 雾效果中的距离因素……231
13.3 相关函数……231

第十四章 交互式图形程序设计……232
14.1 选 择……232
14.1.1 命名几何要素……232
14.1.2 使用选择模式……235
14.1.3 选择缓冲区……235
14.1.4 拾 取……237
14.1.5 有层次的拾取……238
14.2 反 馈……242
14.2.1 反馈缓冲区……242
14.2.2 一个例子……243

14.3 相关函数……248

第十五章 OpenGL 编程技巧……249
15.1 基于MFC的OpenGL编程……249
15.1.1 从应用程序中分离OpenGL程序……249
15.1.2 启动Appwizard……250
15.1.3 MFC程序设计的注意事项……251
15.2 OpenGL 的Visual Basic编程……258
15.2.1 使用OCX……259
15.2.2 在VB5.0中安装使用OpenGL.OCX……260
15.3 OpenGL的Dephi编程……262

第一章 OpenGL 简介

1.1 关于 OpenGL

OpenGL 是一个工业标准的三维计算机图形软件接口。本质上，是一个极其快速并可以方便移植的三维图形和建模库，OpenGL 可以用于创建漂亮的三维图形(与光线跟踪法视觉质量相同)，使用 OpenGL 提供的最大优越性是比光线跟踪法快一个量级。其算法由著名的计算机图形学和动画界领导者 SGI 公司开发并优化。OpenGL 的主要特点是：

①可以在网络上工作，即客户机/服务器型，显示图形的计算机（客户机）可以不是运行图形应用程序的计算机（服务器），客户机与服务器可以是不同类型的计算机，但两者要服从相同的协议。

②可以在多种硬件平台上工作（如个人计算机、工作站），OpenGL 的应用程序有非常好的移植性。

如果在专门为三维图形显示而优化设计的计算机硬件（如配有完全支持 OpenGL 硬件加速的显示卡的计算机）使用 OpenGL，可以大大提高绘图效率，但是仅由软件(称为通用)实现 OpenGL 也是可能的，Microsoft windows NT 和 Windows 95 实现属于这一类。目前，计算机硬件发展很快，有些在几年前还是极其昂贵的专业级显示卡，现在花费几百元钱就可以买到。越来越多的 PC 图形硬件生产商在其产品中加入 3D 加速功能，尽管它更多地受 3D 游戏市场的推动，但同时也推动了基于 Windows 2D 图形加速卡的发展(优化诸如画线、位图填充和操作)，现在不会有人在新机器上仅使用普通 VGA 卡，3D 加速显示卡变得越来越普及。

OpenGL 应用范围很广，不仅用于科研领域中的可视化、三维建模和三维游戏设计，也广泛应用于 CAD 工程、建筑应用软件，甚至用计算机生成 Hollywood 电影中的恐龙。因此，在占有大量市场的操作系统中(如 Microsoft Windows)引入一个工业标准三维图形 API（Application Program Interface，即应用程序接口）有重要意义，三维图形将很快成为消费者和商用软件的一个典型成分。本书正是为满足初学 OpenGL 的读者要求而编写的，全面介绍 OpenGL 的基本概念，由浅入深讲述 OpenGL 的编程技巧。

1.2 OpenGL 的历史

OpenGL 仅出现在几年前，是一个比较新的工业标准。OpenGL 的前身的 SGI 公司的 GL，IRIS GL 是该公司高端 IRIS 图形工作站上三维编程的 API，在 SGI 的计算机上，不仅有经过优化的显示复杂图形的特殊软件，提供异常快速的矩阵变换，硬件支持深度缓冲，还有其他一些特性，但当 SGI 把 IRIS GL 移植到其他硬件平台时出现了问题。OpenGL 是 SGI 为提高 IRIS GL 可移植性而努力的结果，它有 GL 的功能，又是“开放的”，可以适应其他硬件平台的操

作系统。SGI 仍然保留 GL，但不再加强其性能，只是修补一下它的错误。

对 OpenGL 的改进由 ARB 决定，其成员有 SGI、DEC、IBM、Intel、Microsoft 等，几乎所有国际知名的计算机公司。1992 年 6 月，引入了 OpenGL 1.0 版，5 天之后，Win32 开发会上，SGI 演示了运行在 IRIS Indigo 工作站上的 OpenGL，在制造商展厅引起了广泛关注，SGI 和 Microsoft 合作把 OpenGL 引入 Windows NT 的版本。1995 年 12 月，由 OpenGL ARB(Architecture Review Board)通过了 OpenGL 1.1 版，这一版本的 OpenGL 加强了性能并引入了一些新功能。1998 年 3 月，发布了 OpenGL 1.2 版，该版本向上兼容 OpenGL 1.1 版，主要加强了纹理映射和像素处理能力。

1.3 OpenGL 的工作原理

OpenGL 是一种过程图形语言，而不是描述式的，它不描述场景以及显示的方式。实际上，一定外观或效果要通过必需的程序步骤来描述，这些步骤涉及到包含 120 个左右的命令和函数。有高度可移植性的 API 用于绘制诸如三维点、线、多边形等几何要素。另外，OpenGL 支持光照、阴影处理、纹理映射、动画以及其他特殊效果。

OpenGL 不包含任何窗口管理、用户交互、文件 I/O 的函数。 这样，才能使 OpenGL 保持与操作系统无关、高度可移植的特性。每一个主机环境，如 Microsoft 的 Windows 有自己专用函数用于实现把窗口或位图的绘图控制传递给 OpenGL。

OpenGL 初次出现在 Microsoft Windows NT 3.5，其后 Windows 95 也加入了支持 OpenGL 的一系列 DLL 文件。本书主要讲述 Microsoft OpenGL 的基本实现。首先讲述三维图形学基础，然后讲述如何在 Windows NT、Windows 95 环境下编译、联接 OpenGL 程序，再后讲解 Microsoft 提供的可以使 OpenGL 图形 API 与 Microsoft GDI 一起工作的 WGL 函数。

1.3.1 图形体系——软件与硬件

使用 OpenGL 与在 Windows 中用 GDI 绘图有些不一样。事实上，选择当前笔、刷、字体和其他 GDI 实体都不会影响 OpenGL。正如 GDI 用设备描述表控制窗口绘图，OpenGL 用绘图描述表。绘图描述表与设备描述表相关，而设备描述表与 Windows 相联，由此，OpenGL 可以在窗口绘图。

正如前面所说的，在有硬件加速的系统上运行 OpenGL 会大幅度提高绘图命令的效率，PC 图形卡生产商也不断在显示卡中添加对 OpenGL 的支持。另外，正确编写的 OpenGL 应用程序在硬件加速系统和纯软件实现的系统上都可以运行。

图 1-1 说明在 Windows 下硬件加速，包括通常 GDI 加速、Direct Draw 加速和 OpenGL 加速的工作原理。在最左端，应用程序调用 GDI，WINSRV.DLL，Win32 DDI (Device Driver Interface)，Win32 DDI 直接与图形卡驱动程序通讯，在图形卡中完成加速。Direct Draw 用于图形硬件直接存取优化，完全跳过 GDI，除了中间有一个薄硬件提取层(HAL)和对不支持的特性做软件模拟外，直接与硬件打交道。Direct Draw 通常用于游戏， 直接操纵图形内存用于快速 2D 图形和动画。在最右端可以看到，OpenGL 和其他 3D API 通过 3D DDI 调用，3D DDI 为硬件制造厂加速 OpenGL 和诸如 Reality Labs 3D API 游戏。另外，特定 OpenGL 硬件加速(如 GLINT 芯片组)可以安装自己的 OpenGL 客户机驱动程序，用于特定的 *DDI*。

1.3.2 基本实现的限制

除非受硬件支持，Microsoft 的 OpenGL 基本实现有一些限制，不支持向单色打印机或小于 16 色彩色打印机打印 OpenGL 图形，不支持用于各种窗口的硬件调色板，仅有一个调色板，必须在多个应用程序间任选。

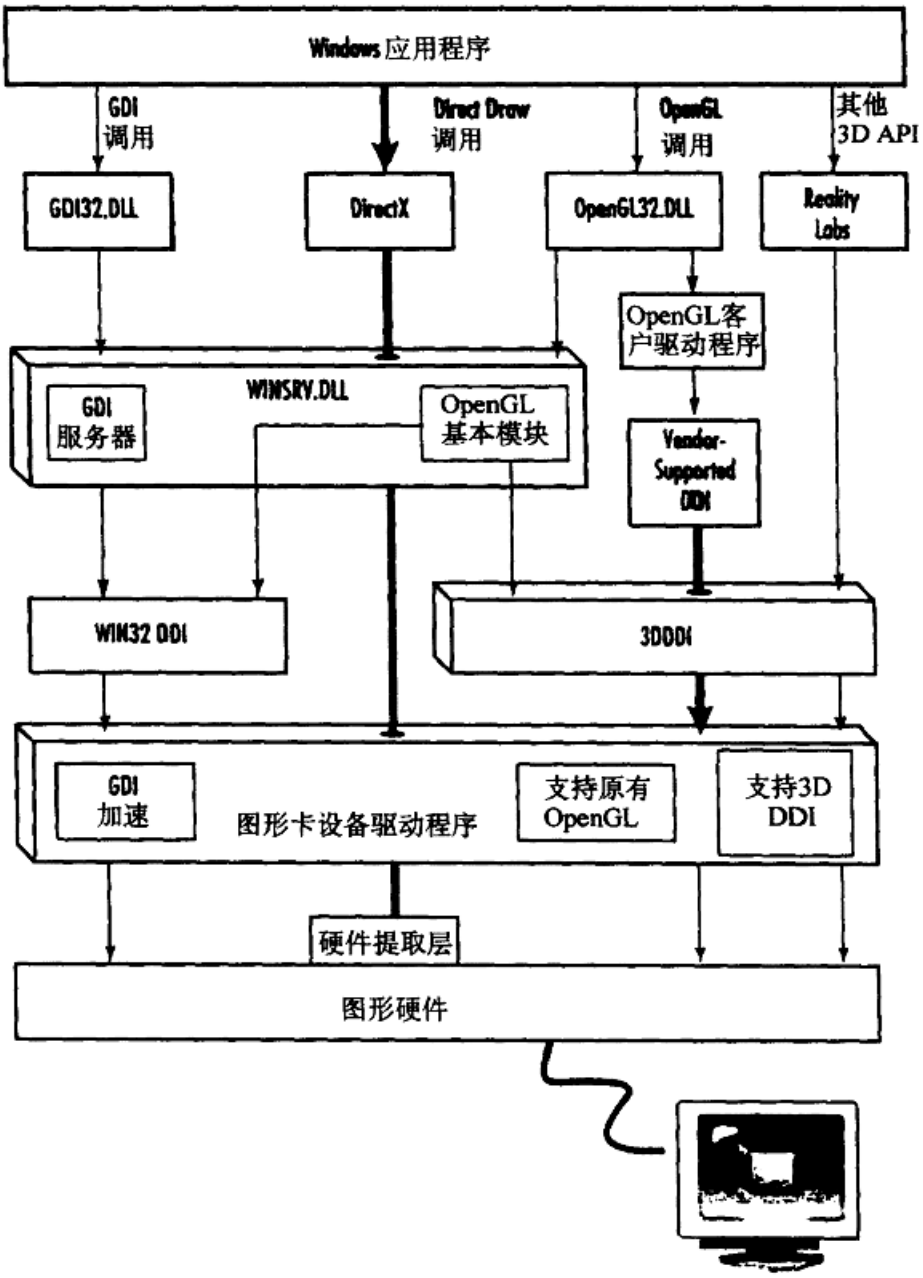


图 1-1 Windows 图形加速的原理图

第二章　三维图形学基础

为方便初学者使用、学习，在使用 OpenGL 创建三维图形之前，要先讲述一些三维图形学术语、三维图形的基本概念和坐标系统，以及如何把计算机屏幕的二维图像理解成三维图形。已经开始使用 OpenGL 有三维图形学经验的读者可以跳过本章。

2.1　三维感受

三维计算机图形实际上是在计算机屏幕上一个有深度(第三维)的二维图像。为真正看到三维物体，要用双眼分别提供物体的图像，每个眼睛接受的图像与视网膜上随时间变化的照片相似。因为得到的角度不同，这两个图像略有差别，大脑把这两个有差别的图像组合起来，就得到一个三维图片。

在图 2-1 物体远离时，夹角 θ 变小，这一三维效应可以通过加大两个图像间夹角进行放大。如果遮住一个眼睛，将无法感受距离远近。

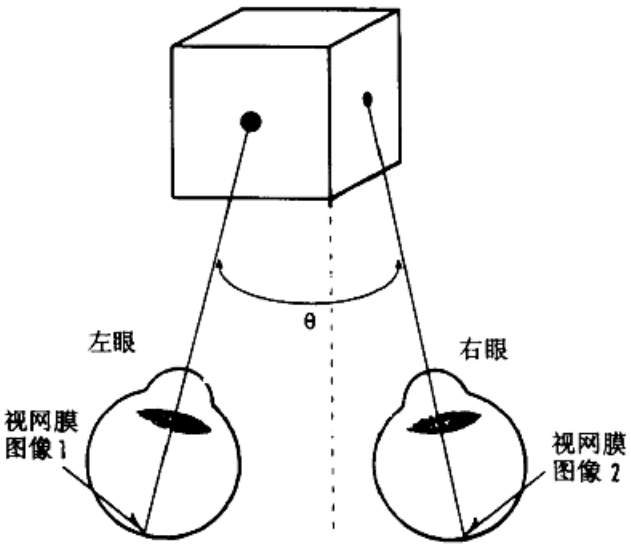


图 2-1　用眼睛观察物体

下面讲述在计算机图形学中是如何感受三维效果的：

●2D+透视投影=3D

为什么遮住一个眼睛后，三维世界并没有变成一个平面世界呢？因为许多三维世界的效果也出现在二维世界，这可以使大脑分辨出深度。对这个问题的最简单的理解就是近大远小，这一效应称为透视，透视与颜色变化、纹理、光照、明暗和颜色强度相互作用在一起才构成对三维图像的感受。

透视本身也可以得到三维外观，图 2-2 是一个简单的方块线框图，即使没有着色和明暗处理，方块仍然有三维物体外观，如果注视方块的时间足够长，方块的前后将换位置，因为大脑会因为图中没有绘制任何表面而混淆了。

●消隐线

图 2-2 包括的信息足可以反映出三维物体的轮廓，但还无法分辨出方块的前面和后面，如果图 2-2 是实体，则看不到方块背面的角点，即使是线框方块，前面的一些线也会遮住后面的部分线框。若在二维图形上模拟这种情况，一定要消掉被前面遮住的那些线，称为消隐线，如图 2-3 所示。

●颜色和明暗处理

图 2-3 仍然不像是真实的物体，方块表面的颜色与背景完全相同，仅能看到物体的前缘，而真实的方块应该有一些颜色或纹理，如木块。在计算机或纸上，若仅给方块着色并在二维上绘制它，得到类似于图 2-4 所示的效果。

如果把边缘都画成不同的颜色，就会体验到三维效果，也可以利用光照给各个侧面有相同颜色的方块做明暗处理，如图 2-5 所示。

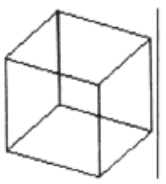


图 2-2　一个简单的方块线框图

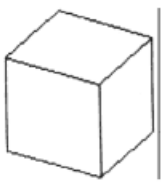


图 2-3　方块线框图消隐

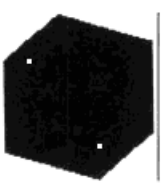


图 2-4　着色的方块线

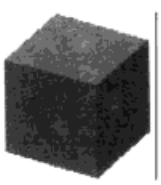


图 2-5　明暗处理的方块

●光照和阴影

正确使用光照效果是 OpenGL 绘制完美的图片过程中一个必不可少的要素。光照对三维物体有两个重要的影响，第一，若以某一角度观察或照亮一个颜色单一的表面，使其看上去有明有暗；其二，由于绝大多数实体不透光，会投下一个阴影。

有 2 个光源影响三维物体：一个是泛光，没有方向，仅仅是能给实体造成明暗效果的均匀光照。泛光使远处边缘看上去暗一些；另一个光源来自称为“灯光”的光源，可以改变实体的明暗，也可以产生阴影。图 2-6 所示为受光源照射的方块所产生的阴影。

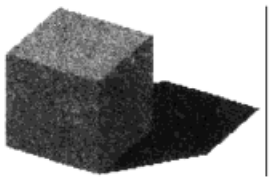


图 2-6　方块的光照和阴影

2.2　坐标系统

既然可以在二维表面(计算机屏幕)感受三维效果，那么如何在屏幕上绘制这些物体呢？在计算机屏幕上绘制点、线、多边形，要根据行、列绘出一个位置。例如，标准 VGA 屏幕由左至右有 640 个像素，由上至下有 480 个像素，屏幕中心的点为(320，240)，即由屏幕左起 320 个像素，由上向下第 240 个像素。

在 OpenGL 中，要为所使用的窗口定义坐标系统，并把所指定的坐标映射到物理屏幕像素上。下面讲述如何在二维绘图中实现，进而把这一原理扩展到三维坐标系统。

1．二维直角坐标

二维图形最常用的坐标系统是直角坐标。由 x,Y 的两个值定义直角坐标，x 坐标为水平方向位置，y 为垂直方向位置的测度。

直角坐标的原点为 x=0，y=0，也可以写成坐标对(0,0)形式，图 2-7 描述的即是二维直角坐标，x 轴与 y 轴互相垂直。可以用不同的窗口映射方法，使 OpenGL 对用户定义的坐标在绘图时有不同的理解。本书后面将讲述这些真实空间坐标向窗口坐标的不同映射方法。

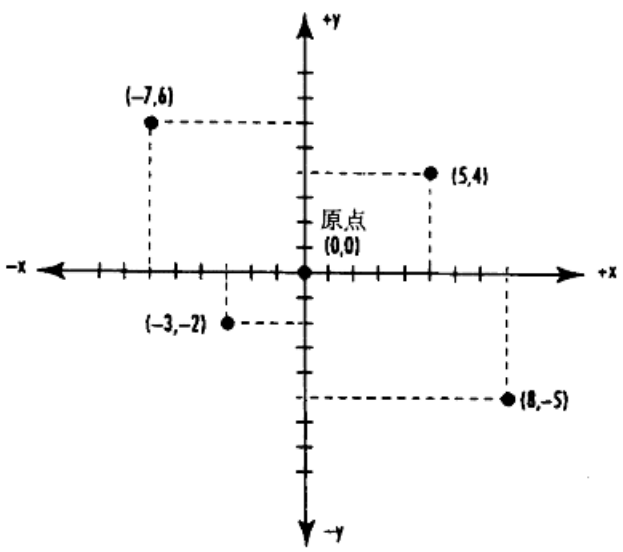


图 2-7　二维直角坐标

2．坐标裁剪

窗口长度的单位是像素，在开始绘制点、线、多边形之前，用户要告诉 OpenGL 如何把指定的坐标对变成屏幕坐标，这可由指定窗口上直角坐标空间区域来完成，这一区域称为裁剪区。在二维空间，裁剪区是窗口内 x,y 的最大、最小值；另一个方法是定义相对于窗口的原点位置，图 2-8 绘出这两个常用的裁剪区。

第一个例子中(图 2-8 左)，窗口内 x 由左到右的取值范围为 0 到 +150，y 由下到上的取值范围是 0 到+100，屏幕的中心坐标为(75，50)，第二个例子的 x 取值为-75 到+75，y 取值范围-50 到+50，屏幕的中点为(0,0)。也可以用 OpenGL 函数(或普通 GDI 绘图的 Windows 函数)把坐标系统颠倒过来。应该注意的是，缺省的 Windows 窗口是+y 轴由窗口顶部指向底部。尽管写文本是由上至下有益处，但绘图时这个缺省映射不方便。

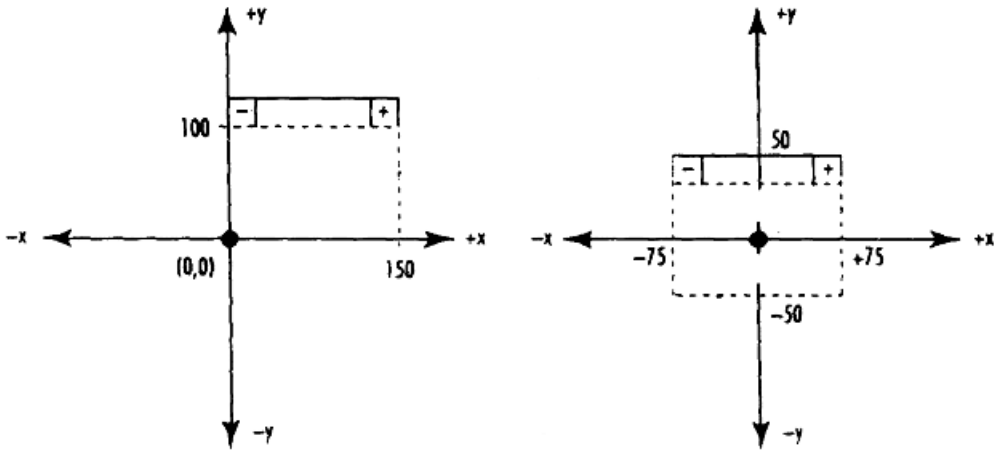


图 2-8　两个裁剪区

3．视见区——把窗口变为三维

以像素为长度单位的窗口尺寸只有在特殊情况下，裁剪区的宽、高才完全一致，因此要把逻辑上的直角坐标映射到物理上的屏幕像素坐标，这一映射可由设定视见区完成，视见区是窗口中用于绘制的裁剪区域。把裁剪区映射到窗口的某一区域，通常把视见区设为整个窗口，当然，也可以是窗口的一个部分。

图 2-9 给出一个 300×200 像素的窗口，视见区为整个窗口大小，若把裁剪区设为 x 轴的范围 0 到 150，y 轴 0 到 100，在视见窗口上，逻辑坐标会被映射到一个更大的屏幕坐标系统。逻辑坐标系统的一个单位长度与窗口的物理坐标系统 2 个单位长度相匹配。

若视见区与裁剪区相匹配，视见窗口大小仍为 300×200，那么视见区仅为窗口的左下角，如图 2-10 所示。

可以用视见区放大、缩小窗口上的图像。如果把裁剪区设置得比窗口还要大，则仅能显示一部分裁剪区。

4．绘图要素

无论是二维还是三维空间，所绘制的物体都是由更小的几何形状组合成的，这些小单元称为几何要素。几何要素是诸如点、线、多边形等二维表面，把它们装配到三维空间就可以创建三维物体，如图 2-5 的三维方块是由 6 个二维正方形构成的，几何要素的角点称为顶点。在第五章中将介绍 OpenGL 几何要素及其使用。

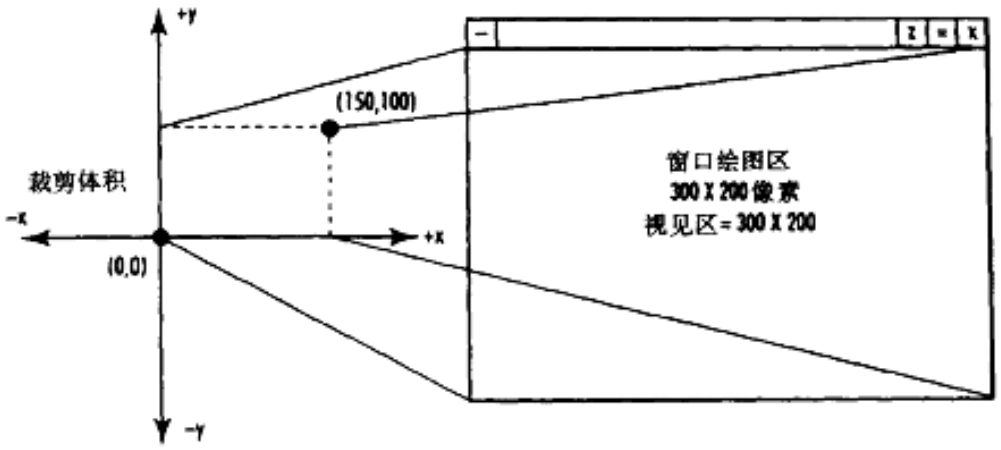


图 2-9　视见区尺寸为裁剪区的两倍

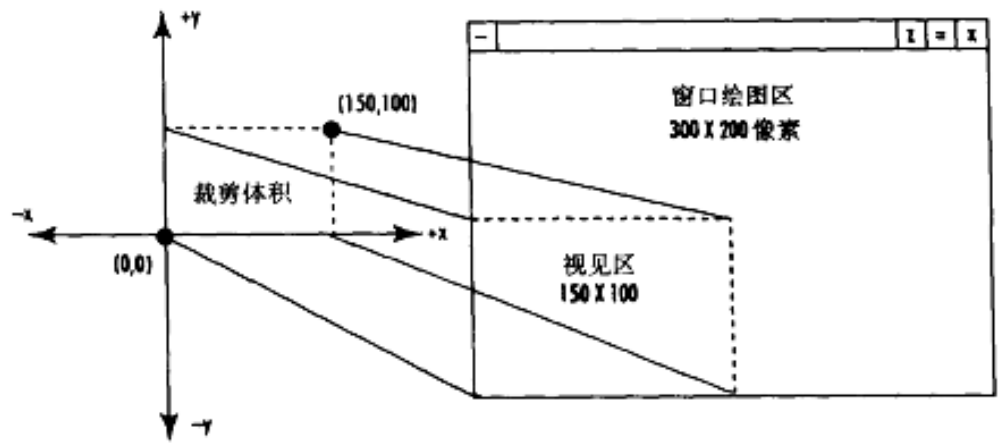


图 2-10　视见区与裁剪区面积相同

5. 三维直角坐标

把二维坐标系统扩展到三维要加上深度坐标，图 2-11 为加上 z 轴的直角坐标系统，z 轴垂直于 x, y 轴，z 轴的正方向是垂直于屏幕指向观察者的，在三维坐标系统中定义一个位置要用 x, y, z 三个坐标。

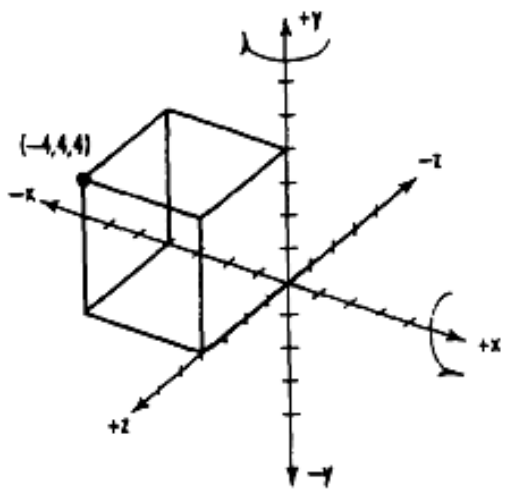


图 2-11　三维直角坐标系统

2.3 投影三维的实质

尽管在三维空间可以用直角坐标定义位置，但屏幕上的像素是二维的，OpenGL 是如何把这些三维坐标转换成屏幕上的二维坐标的呢？坐标转换是通过三角函数和简单矩阵变换完成的。有关内容将在本书第六章中讲述。然而，用 OpenGL 创建图形时，不必对这些数学公式有更深入理解，只需理解称为所谓“投影”的概念。把三维坐标投影到二维屏幕上，如图 2-12 所示为把一个屋子投影到一块玻璃上。通过定义投影，用户可以指定显示在窗口中的剪裁体积。在 OpenGL 中有两大类投影方式：正交投影与透视投影。

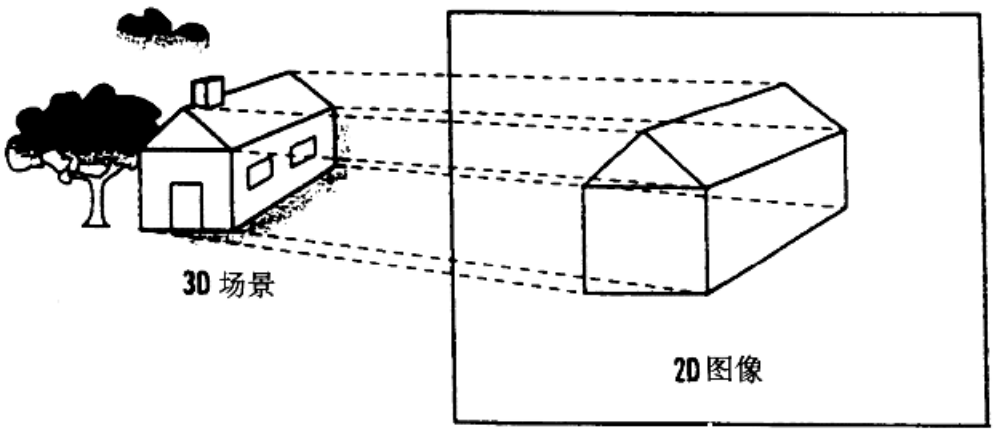


图 2-12 投影示意图

1. 正交投影

正交投影或称平行投影，这种投影方式是指定一个方形或矩形裁剪体积使用，在这个裁剪体积外的物体被丢掉，被投影物体的尺寸不变。这类投影方式常用于建筑设计或 CAD。

正交投影的裁剪体积由远、近、左、右、顶、底裁剪平面构成，在这一体积内的物体被投影到屏幕上成为二维图像。

2. 透视投影

透视投影是另一个常用的投影方式，它考虑物体近大远小的视觉效果。其裁剪体积称为截头锥体，越靠近取景体积前面的物体越接近其原有尺寸，反之亦然。这类投影在三维动画和模拟时更逼真。

2.4 本章小结

本章介绍了三维图形学基础，讲述了为什么要从不同角度感受真实三维空间，讲述了投影、消隐、着色、明暗处理、光照技巧，以及 OpenGL 把三维图形投影到二维屏幕的两种方法。

本章有意略去了 OpenGL 创建这些效果的细节。本书后面章节，将讲述如何使用这些技巧并最大限度地开发 OpenGL 的性能。

第三章　使用 AUX 库函数

本章首先介绍 OpenGL 的工作原理，再介绍给变量和函数命名的规则，如果读者已经开始写一些程序，就可以跳过第一部分，而直接学习 AUX 库函数使用部分的内容。

3.1　OpenGL API

与大家熟悉的计算机高级语言，如 C、FORTRAN 等不同，OpenGL 不是一门计算机编程语言，而仅是一个 API (Application Program Interface，即应用程序接口)。如果提到一个程序是基于 OpenGL 的或是 OpenGL 应用程序，那么这个程序是用某一编程语言写成的，只是在其中调用了 OpenGL 的一些库函数。OpenGL 的库函数有在各种编程语言下的定义，如 C 语言、FORTRAN 等。当然，可以把多个图形软件包结合起来，由 OpenGL 完成一些特定的任务，而用特定环境下的图形软件，如 Windows GDI 完成其他工作。

作为一个 API，OpenGL 库函数符合 C 语言调用规则。这就是说，用 C 语言编写的程序可以比较方便地调用 API。其原因在于：一方面，函数本身用 C 语言编写；另一方面，C 语言提供了一系列调用汇编语言或其他语言的中间函数。本书多数的程序用 C 或 C++编写，运行在 Windows NT、Windows 95 环境下。C++程序可以与 C 语言相同方式调用 C 函数和 API。在 Microsoft Visual Studio 环境下的 FORTRAN 可以很方便地使用 OpenGL 模块，并实现与 Visual C++的混合编程。其他编程语言，如第四代编程语言(称为“4GLs”)类，如 Visual Basic 也能够通过 OCX 控件调用 OpenGL（本书第十五章将进行详细讨论)。

3.1.1　OpenGL API 的构成

OpenGL API 由三个不同的库函数构成：

①辅助库 AUX，也称为“工具箱”库函数（glaux.lib)。该库函数的声明在 glaux.h 头文件中，这一库函数的内容并不是 OpenGL 定义的内容，而是为调用 OpenGL 函数提供的与平台独立框架的工具箱。此库函数的前缀为“aux”。

②OpenGL 核心库（opengl32.lib)。这些函数由 OpenGL ARB 定义，是 OpenGL 的核心函数。其头文件为 gl.h，前缀为“gl”。

③OpenGL 应用库函数（glu32.dll)，头文件为 glu.h。这个库包括的函数可以使用户更方便地编写程序、完成任务，如绘制球、盘、圆柱。应用库函数实质上是用 OpenGL 命令写成的，可以在任意支持 OpenGL 的平台上使用。这些函数的前缀是“glu”。

值得注意的是，在运行 OpenGL 应用程序时，Windows 系统所在目录的 system 子目录中一定要有 opengl32.dll、glu32.dll。

本章主要介绍 AUX 库函数、OpenGL 基础以及 gl 库的一些命令。

3.1.2 OpenGL 数据类型

为使 OpenGL 源代码可以方便地从一个平台移植到另一个平台，OpenGL 定义了自己的数据类型。这些数据类型与常用的 C 语言数据类型一一对应。不同的编译器和环境又有各自用于不同 C 变量的大小及内存设置的规则。用 OpenGL 定义变量类型，可以不受这些类型变化的影响。

表 3-1 给出了 OpenGL 数据类型和相应 32 位 Windows 环境下 C 语言数据类型及后缀。许多 OpenGL 函数名也用这些后缀。

表 3-1 OpenGL 数据类型

OpenGL 数据类型	内部表示	C 定义	C 后缀
GLbyte	8 位整数	Singed char	b
GLshort	16 位整数	Short	s
GLint, GLsizei	32 位整数	Long	l
GLfloat, GLclampf	32 位浮点数	Float	f
GLdouble, GLclampd	64 位浮点数	Double	d
GLubyte, GLboolean	8 位无符号整数	Unsigned char	ub
GLushort	16 位无符号整数	Unsigned short	us
GLuint, GLenum, GLbitfield	32 位无符号整数	Unsigned long	ui

从表 3-1 可以看出，所有数据类型均以“GL”开头，代表 OpenGL 的数据类型，大多数与相应 C 语言数据类型相同。有些在数据类型前加“u”的表示无符号数据类型，如 ubyte 为无符号字节。有时给出一个描述得更清楚的名字，如 size 表示一个有长度或深度的值。GLsizei 表示尺寸参数为整数的 OpenGL 变量。clamp 用于颜色组分，代表 Color amplitude，即颜色幅值。这一数据类型的 f，d 后缀分别表示浮点、双精度数据类型。GLboolean 变量标识是 True 或 False 逻辑条件。GLenum 为枚举型变量。GLbitfield 用于包含二进字位的变量。

指针和数组没有特殊定义，例如，包含 10 个 GLshort 变量的数组可以声明为

```
GLshort shorts[10];
```

有十个指向 GLdouble 变量的指针可以声明为：

```
GLdouble *doubles [10];
```

另外，一些指针类型对象用于 NURBS 和二次曲面，有关内容将在本书后面的章节中详细介绍。

3.1.3 函数命名规则

OpenGL 函数名称遵循一定的命名规则。命名规则使用户知道该函数属于哪个库，函数的参数及其数据类型。所有函数都有一个词根代表着相应的 OpenGL 命令，如 glColor3f 函数的词根为 color，是 OpenGL 设置颜色的命令，gl 前缀代表这个命令在 gl 库中，3f 后缀代表该函数有 3 个浮点参数。按照相同的约定，所有 OpenGL 函数的格式都可以写成：

<库函数前缀><命令词根><可选择的参数数目><可选择的参数类型>

3.2 AUX 库函数

AUX 库函数是为简化学习和编写 OpenGL 程序而创建的。借助于这个库函数，用户编写的 OpenGL 应用程序可以不必针对某一个特定的环境。这样的程序，无论是在 Unix、Windows，还是其他平台环境下，都可以正确运行。虽然使用 AUX 库函数可以很容易实现用户的想法，但是，缺少具体的 GUI 特征，也限制了用这一库函数构造实用的应用程序。

几乎每一个 OpenGL 实现都有一套 AUX 核心函数。用户使用这些函数，可以处理创建、控制窗口和用户输入事件。AUX 库中还有一些函数可以绘制复杂的三维线框图或实体图。用 AUX 库函数可以绘制相当复杂的物体，而把一个平台的应用程序移植到其他环境时，只需重新编译就能使用。

除了 AUX 核心函数外，实现 AUX 库函数的环境也创建了针对完成特定系统进行操作，如交换缓冲区和读入图像的一些帮助函数。用户的源代码越依赖这些附加 AUX 库函数，所编写的 OpenGL 程序可移植性就越差。尽管如此，初学 OpenGL 的读者充分利用这些函数又可以不必学习太多 Windows 编程细节，而在屏幕上绘制出漂亮的场景。

值得注意的是，AUX 库函数并没有包含绘制所有三维物体的函数。

●与平台无关性

OpenGL 是一个强大的、复杂的创建三维图形的 API。有 300 多个命令，涉及从设定材料颜色、反射特性到进行旋转和复杂坐标变换等各个方面，但没有一个函数与管理窗口相关，也没有函数处理键盘输入，鼠标交互作用。因为设计 OpenGL 的主要目的是与平台无关。创建、打开窗口在不同的平台下是不同的。

另一个与平台相关的问题是在不同操作系统的环境下对键盘、鼠标输入事件的处理，而不同平台的处理方法不同，所以 OpenGL 的与平台无关性是以牺牲 OS 和 GUI 函数为代价的。

●AUX=平台 I/O

AUX 库函数是为提供一个不必学习特定操作系统和用户接口细节的工具箱而创建的。为此，AUX 提供了创建窗口、读入鼠标、键盘事件的基础函数。AUX 库函数可以在内部为这些函数利用当前环境的 API，从而使这些函数在所有平台上保持不变。

AUX 库函数仅包含几个管理窗口、处理输入事件的函数，但省掉了通过 Windows API 在 C 或 C++中实现这些管理功能的麻烦。该库函数也有绘制 如球、方块、环甚至茶壶等相对复杂三维物体的函数。可以很轻松地用 AUX 库函数显示一个窗口并执行一些 OpenGL 命令。

尽管 AUX 并不真正属于 OpenGL 定义，但每一个可移植的平台都有该库函数，当然，Microsoft Windows 也不例外。AUX 库函数的源代码可以在 Microsoft 的 Win32 SDK 光盘上免费获得。

3.3 解剖一个 OpenGL 小程序

本节将结合一个非常短小的 OpenGL 程序(程序 3-1)，说明如何使用 AUX 库函数编写应用程序。虽然这个程序所具有的功能仅仅可以打开一个窗口，却是一个比较完整的程序。它是本书在介绍了一些图形学基本概念后，给出的第一个 OpenGL 程序。在进一步掌握 OpenGL

的函数后，以此作为基础，不断对其进行丰富、完善，可以编写出功能更强大的应用程序。

程序 3-1

```
// The shortest OpenGL program possible

#include <windows.h>     // Standard Window header required for all programs
#include <conio.h>       // Console I/O functions
#include <gl\gl.h>       // OpenGL functions
#include <gl\glaux.h>    // AUX Library functions

void main(void)
    {
    // These are the AUX functions to setup the window
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,250,250);
    auxInitWindow("My first OpenGL Program");

    // The are the OpenGL functions the do something in the window
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    glFlush();

    // Stop and wait for a keypress
    cprintf("Press any key to close the Windcw\n");
    getch();
    }
```

创建这个程序时，要设定编译、联接的选项，即用户要创建一个 Win32 控制台应用程序并在联接选项中选择 glaux.lib、opengl32.lib 两个库函数。

运行这个程序时，创建一个标准 GUI 窗口，标题为“My first OpenGL Program”，把背景设为蓝色。在控制台窗口显示“Press any key to close the window”。GUI 窗口不对键盘、鼠标事件响应，而控制台窗口在等待键入任一键后结束程序。这个程序的功能很不完善，OpenGL 窗口不能移动、改变大小，甚至不能刷新。若用另一窗口遮住该窗口再移开，会发现绘图变黑了。

这个简单程序包含了 3 个 AUX 库函数和 3 个 OpenGL 函数，下面将逐行分析该程序，然后引入另外一些函数，以改进这个程序的功能。该程序的构成为：

①包含文件

```
#include  windows.h>
#include <conio.h>
#include <gl\gl.h>
#include <gl\glaux.h>
```

这些包含文件定义程序中使用的函数类型，其中 windows.h 头文件是 Windows GUI 应

用程序必需的。尽管这是一个控制台模式的程序，但 AUX 库函数要创建 GUI 窗口并在其中绘图。头文件 conio.h 用于控制台 I/O，因为 cprintf( )要显示信息，getch( )等待敲击键盘。gl.h 定义前缀为“gl”的 OpenGL 函数；glaux.h 包含 AUX 库必需的函数。

②主程序

```
void main(void) {
```

控制台模式 C 和 C++程序通常以 main( )函数开始。与在 GUI 应用程序中可以创建控制台窗口一样，用控制台应用程序在 Win32 中也可以创建图形窗口。只是这些细节均包容在 AUX 库中。本书将在下一章详细介绍在 Windows 环境如何编写 OpenGL 程序。

③显示模式：单缓冲区

```
auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
```

使用 AUX 库函数创建窗口的显示模式，在这个程序中用单缓冲区窗口(AUX_SINGLE)并使用 RGBA 颜色模式(AUX_RGBA)。单缓冲区窗口意味着所有绘图命令均在这个屏幕显示的窗口执行，而与之对应的双缓冲区窗口中，绘图命令则是在一个脱离屏幕的绘制场景执行，然后快速切换到屏幕上的窗口。双缓冲区窗口通常用于产生动画效果。这些内容在本章后面内容中做介绍。RGBA 颜色模式表明分别用指定红、绿、蓝组分的强度定义颜色，是 OpenGL 定义颜色的模式之一(请参见本书第七章的内容)。

④定位窗口

在设置显示模式后，需要确定窗口的位置和大小。

```
auxInitPosition(100,100,250,250);
```

这些参数代表窗口左下角的位置以及窗口的宽、高。在本程序中，把窗口左下角放置在(100,100)，窗口的宽、高均为 250，该函数的定义为：

```
auxInitPosition(GLint x, GLint y, GLsizei width , GLsizei height);
```

GLint、GLsizei 数据类型定义为整数，x 值为由左向右屏幕的像素数，y 为由上至下屏幕的像素数目，这是 Windows 缺省的映射方式，而 OpenGL 缺省方法是 y 轴由下向上为正方向，x 坐标的映射相同。这两种映射方式的差异应该引起读者足够的重视，在设计 OpenGL 程序时，要时刻注意到这一点。两种坐标映射的示意图见图 3-1。

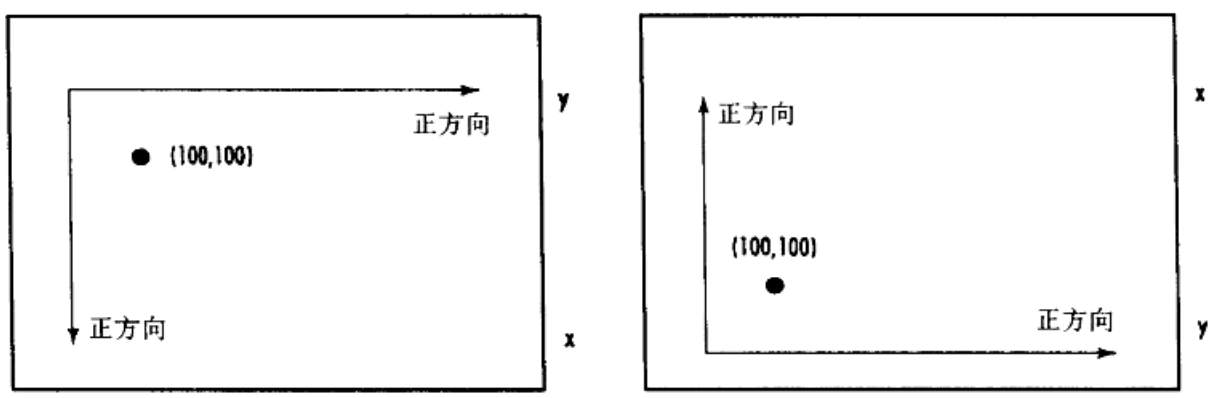


图 3-1　两种坐标映射的示意图

⑤创建 OpenGL 窗口

```
auxInitWindow("My first OpenGL Program");
```

创建一个标题为“My first OpenGL Program”的窗口。这个函数的唯一参数是窗口的

标题。如果程序到此结束，这个程序的运行过程将是首先创建一个有指定标题的空窗口，再关闭窗口，最后结束。getch( )函数保留了窗口，但没有任何有用的显示。

⑥清除窗口

调用上述三个 AUX 库函数已经完成了对 OpenGL 要使用的窗口初始化、创建工作。由此开始，所有的 OpenGL 命令和函数调用操作都针对这个窗口。

```
glClearColor(0.0f, 0.0f, 1.0f, 1.0f);
```

是本书的第一个 OpenGL 函数调用。它设置清除窗口所用的颜色，即设置所创建窗口的背景颜色。在程序 3-1 中，背景颜色为蓝色。该函数的定义为：

```
glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
```

GLclampf 对多数的 OpenGL 实现定义为浮点数。 在 OpenGL 中，颜色是红、绿、蓝组分混合而成的，每个组分的取值范围是 0.0 至 1.0。如果各组分取该范围内所有值，在理论上可以构成无穷多个颜色。在系统内部，OpenGL 把颜色表示成一个 32 位值，最多可达 4294967296 种颜色(有时称为真彩色)，因此,0.0 到 1.0 取值的有效步长大约为 0.00006。

glClearColor( )的最后一个参数是 alpha，用于混合和透明等特殊视觉效果，将在本书第十三章中介绍。

⑦清除操作

尽管已经为 OpenGL 设置用于消除的颜色，但是真正完成清除操作要由下面的函数

```
glClear(GL_COLOR_BUFFER_BIT);
```

来完成。glClear( )函数可以消除一个缓冲区或几个缓冲区的组合。在上面的程序中，仅清除了颜色缓冲区。颜色缓冲区是所显示图像在内存的存贮区，用 glClear 函数清除缓冲区也消掉了窗口上的图形。有效使用缓冲区是 OpenGL 的一个重要特征，有关内容将在第十二章中详细介绍。

⑧完成命令序列

程序 3-1 的最后一个 OpenGL 函数调用为：

```
glFlush( );
```

使所有尚未执行的 OpenGL 命令，即 glClearColor( )和 glClear( ) 得到执行。在内部，OpenGL 采用一个绘图管道机制按顺序处理这些命令，从而大大提高了性能，尤其在绘制复杂物体时更能体现出该机制的优势。因为绘图对一给定的命令与低速的图形硬件打交道少，使 OpenGL 得到加速。与此类似的是，在首次引入 Win32 时，在 Windows NT 环境下，也采用相同的概念提高 Windows GDI 的图形性能。在这个小程序中，glFlush( )函数使 OpenGL 执行已有的绘图命令而不是继续等待任何其他命令。

本程序最后一段为：

```
// Stop and wait for a keypress
cprintf("Press any key to close the Window\n");
getch();
}
```

在控制台窗口显示信息，并在击任一键后停止执行程序，关闭窗口。

程序 3-1 给出了用 AUX 库函数创建窗口的最基本知识，也给出了如何定义颜色和清除窗口。下面将添加一些 AUX 库函数和 OpenGL 命令完善这个程序的功能。

3.4 用 OpenGL 绘制几何形状

程序 3-1 建立了一个蓝色背景的空窗口。程序 3-2 中不再用 getch( ) 函数终止执行程序，也就不必在包含文件中加入 conio.h 头文件。该程序要完成在窗口中绘图、移动窗口、改变窗口大小等操作，使之有一般 Windows 窗口的功能。下面具体分析这个程序的结构和功能。

程序 3-2

```
// A more friendly OpenGL program

#include <windows.h>     // Standard header for Windows
#include <gl\gl.h>       // OpenGL library
#include <gl\glaux.h>    // AUX library

// Called by AUX library to draw scene
void CALLBACK RenderScene(void)
     {
     // Set clear color to Blue
     glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

     // Clear the window
     glClear(GL_COLOR_BUFFER_BIT);

     // Set current drawing color to red
     //           R      G      B
     glColor3f(1.0f, 0.0f, 0.0f);

     // Draw a filled rectangle with current color
     glRectf(100.0f, 150.0f, 150.0f, 100.0f);

     glFlush();
     }

 void main(void)
     {
     // AUX library window and mode setup
     auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
     auxInitPosition(100,100,250,250);
     auxInitWindow("My second OpenGL Program");

     // Set function to call when window needs updating
```

```
    auxMainLoop(RenderScene);
    }
```

①绘图函数

```
// Called by AUX library to draw scene
void CALLBACK RenderScene(void)
```

把在窗口中实际绘图的语句都移到这个子程序中，本书后面的程序中也遵循这一约定。值得注意的是，在函数声明中出现 CALLBACK 是必需的，因为只要窗口发生变化，AUX 库函数就要调用有该标志的子程序。

②画一个矩形

画矩形的程序段如下：

```
    // Set current drawing color to red
    //          R      G      B
    glColor3f(1.0f, 0.0f, 0.0f);

    // Draw a filled rectangle with current color
    glRectf(100.0f, 150.0f, 150.0f, 100.0f);
```

glColor3f( )函数为后面绘图命令选定一个颜色。glRectf( )函数用当前颜色画一个填充矩形。glColor3f( )函数选择颜色的方法与 glClearColor( )相同，但不用定义 alpha 组分，该函数的定义如下：

```
    glColor3f(GLfloat red, GLfloat green, GLfloat biue);
```

glRectf( )函数的参数为4个浮点值，分别定义了矩形的左上角(x1, y1)和右下角(x2, y2)的坐标。函数的定义如下：

```
    glRectf(GLfloat x1, GLfloat y1, GLfloat x2, GLfloat y2);
```

③初始化

程序 3-2 的主程序如下：

```
void main(void)
    {
    // AUX library window and mode setup
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,250,250);
    auxInitWindow("My second OpenGL Program");

    // Set function to call when window needs updating
    auxMainLoop(RenderScene);
    }
```

与程序 3-1 相同，三个以 auxInit 开头的函数调用设置并显示所要绘图的窗口。auxMainloop( )的参数为执行绘图子程序 RenderScene( )的指针。当第一次显示窗口，移动窗口、改变窗口大小，被其他窗口遮挡，再移动该窗口时，都要调用 RenderScene( )这个回调函数。

程序 3-2 可以在蓝色窗口中心显示一个红色方块，如图 3-2 所示。因为方块的位置固定，若把窗口放大，红色方块仍然处在窗口左下角；若窗口过小，则不能完整显示出该方块。

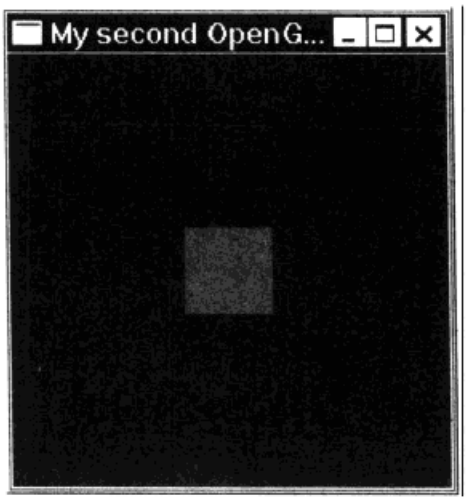


图 3-2　程序 3-2 的屏幕显示

④缩放窗口

几乎在所有的窗口环境下，用户都可随时改变窗口大小。此时，通常要重画窗口所显示的内容，以适应新的窗口尺寸。有时希望用较小的窗口裁剪图形或用较大窗口显示图形原有的大小，有时要图形大小随窗口尺寸变化，使小窗口显示的图片尺寸小，但是完整，大窗口内的图片大，但是与原图相似。

⑤设置视见区和裁剪体积

在本书第二章中，已经讲述了视见区和裁剪体积是如何影响计算机屏幕上二维窗口的坐标范围及二维、三维图形的大小的。下面结合程序 3-2，看一看在 OpenGL 编程时，如何设定视见区和裁剪体积坐标。

当用 auxInitPosition (100, 100, 250, 250)创建窗口时，在缺省情况下,AUX 库函数把视见区设为与窗口尺寸完全相同。裁剪体积在 xy 坐标的第一象限，尽管程序 3-2 绘制的是一个二维矩形，实际上也是在三维空间完成绘图，即 glRect( )函数绘制的矩形在 z=0 的 xy 平面上。

一旦窗口尺寸变化，视见区和裁剪体积就要重新定义，否则，会出现无论窗口尺寸怎样变化，坐标系统到屏幕坐标的映射不变的情形，从而导致绘图区域占整个窗口的比例不正确。

在不同的环境下，检验和处理窗口尺寸变化的方法也不同。在 OpenGL 中，可以通过调用 AUX 库函数提供的 auxReshapeFunc( )函数，使窗口尺寸变化时，AUX 库函数回调一个函数进行处理。这一函数可以写成如下形式：

```
void CALLBACK ChangeSize(GLsizei w, GLsizei h);
```

其中 w，h 分别为改变后窗口的宽、高。本书后面所举例子中，也用这个子程序处理窗口尺寸变化的情况。

在窗口尺寸变化时，Changesize( )函数接受新窗口的宽、高，可以借助 glViewport( )和 glOrtho( )两个函数来改变坐标系统到屏幕坐标的映射。程序 3-3 是考虑到窗口尺寸变化后对程序 3-2 的修改。下面仅给出改变了的 main( )函数和新函数 ChangeSize( )。程序 3-3 的屏幕显示如图 3-3a 所示。

程序 3-3

```
// Scaling an OpenGL Window.

#include <windows.h>     // Standard Windows header
#include <gl\gl.h>       // OpenGL Library
#include <gl\glaux.h>    // AUX Library

// Called by AUX Library when the window has changed size
void CALLBACK ChangeSize(GLsizei w, GLsizei h)
     {
     // Prevent a divide by zero
     if(h == 0)
          h = 1;

     // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

     // Reset coordinate system
     glLoadIdentity();

     // Establish clipping volume (left, right, bottom, top, near, far)
    if (w <= h)
          glOrtho (0.0f, 250.0f, 0.0f, 250.0f*h/w, 1.0, -1.0);
    else
          glOrtho (0.0f, 250.0f*w/h, 0.0f, 250.0f, 1.0, -1.0);
     }

// Called by the AUX Library whenever the window
// needs to be updated
void CALLBACK RenderScene(void)
     {
     // Set background color to blue
     glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

     // OpenGL or AUX Library drawing code
     glClear(GL_COLOR_BUFFER_BIT);

     // Draws a Red Rectangle
     glColor3f(1.0f, 0.0f, 0.0f);
     glRectf(100.0f, 150.0f, 150.0f, 100.0f);
```

```
    glFlush();
    }

void main(void)
    {
    // Setup and initialize AUX window
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,250,250);
    auxInitWindow("Scaling Window");

    // Set function to call when window changes size
    auxReshapeFunc(ChangeSize);

    // Set function to call when window needs updating
    auxMainLoop(RenderScene);
    }
```

(a) (b)

图 3-3 缩放窗口前后

当改变窗口尺寸时，方块的尺寸也发生变化，如图 3-3b 所示，值得注意的是，无论窗口尺寸如何变化，窗口中的矩形始终为正方形。

⑥定义视见区

在 ChangeSize( )函数中，首先调用 glViewport 函数传递最新窗口宽、高值。glViewport( )函数定义为：

```
void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);
```

参数 x，y 为窗口内视见区的左下角坐标，缺省的 x，y 值均为零。用户可以在同一个

窗口内定义几个视见区，在不同区域绘制多个图。视见区定义了在窗口上用于绘制 OpenGL 图形的以屏幕坐标为单位的区域，如图 3-4 所示。定义了视见区后，当前裁剪体积被映射到新的视见区。若所定义的视见区小于窗口，图形就要缩小。

⑦定义裁剪体积

在 ChangeSize( )函数的最后重新定义了裁剪体积，使长宽比保持一致。长宽比的定义为垂直方向与水平方向单位长度的比值。当长宽比为 1.0 时，两个方向长度单位一致。

若定义的视见区不是正方形，即映射到正方形裁剪体积上图像要变形。例如在矩形窗口上绘制逻辑坐标上的正方形，将得到与窗口长宽比一致的矩形，不会得到正方形。

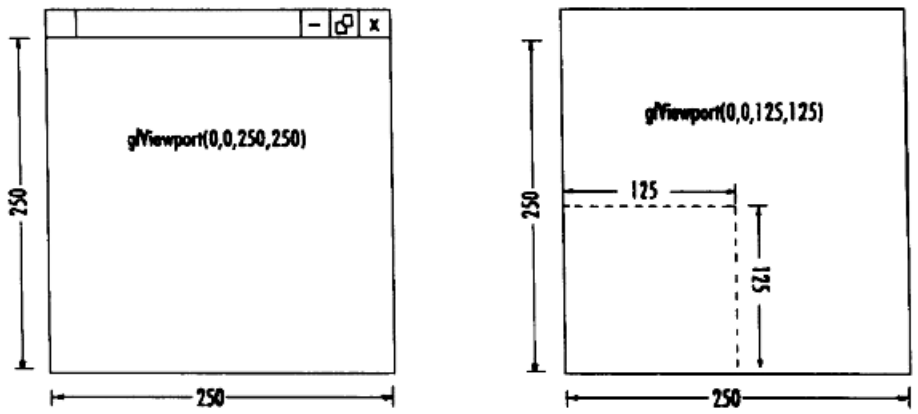


图 3-4　视见区与窗口间的映射

本程序，在裁剪体积中采用正交投影。创建这一投影的 OpenGL 命令是 glOrtho( )函数，其定义为：

```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
             top, GLdouble near, GLdouble far);
```

在三维直角坐标空间，左、右值分别定义了 x 方向最小、最大值；底、顶值对应 y 轴，近、远值对应 z 轴，通常负值远离观察点，见图 3-5。

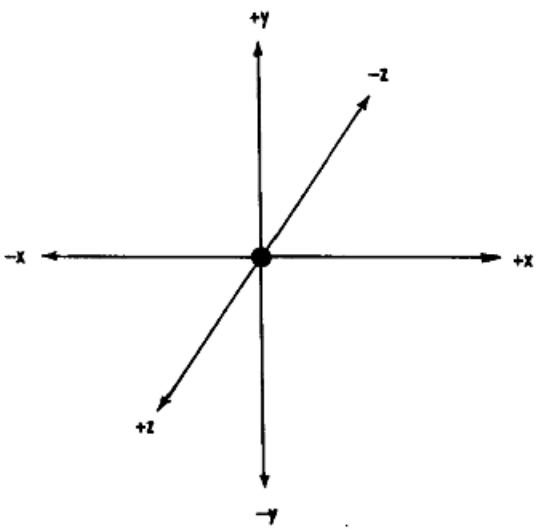


图 3-5　三维直角坐标空间

在调用 glOrtho( )函数前，应该首先调用 glLoadIdentity( )，因为 glOrtho( )并不能建立裁剪体积，仅仅是改动已有的裁剪体积。它用该函数参数描述的矩阵来替代当前裁剪体积描述的矩阵。有关矩阵操作和坐标变换的内容见第六章。glLoadIdentity( )可以在做任何矩阵操作之前，把坐标系统复位，即三维直角坐标空间三个方向的单位长度均相同。如果不做复位，每次调用 glOrtho( )，其后调用 glOrtho( )都会进一步破坏建立好的裁剪体积。

⑧如何使正方形保持原形状

下面的程序段可以保持原有形状：

```
if (w <= h)
    glOrtho (0.0f, 250.0f, 0.0f, 250.0f*h/w, 1.0, -1.0);
else
    glOrtho (0.0f, 250.0f*w/h, 0.0f, 250.0f, 1.0, -1.0);
```

修改裁剪体积，可以使宽度不大于高度时左边保持 x=0，右边为 250，否则右侧要根据长宽比发生变化；高度不大于宽度时，底部 y=0，顶部为 250，否则，顶部位置要根据长宽比变化。这样可以使正方形无论在什么形状的窗口下都仍然为正方形。

3.5 用 AUX 做动画

到目前为止，本章已经讲述了如何用 AUX 库创建窗口、用 OpenGL 命令绘图等内容。现在讲述 AUX 库函数实现动画的方法。下面给出一个用 AUX 做简单动画的程序。在程序 3-4 中，上一个例子绘制的正方形在窗口的边框间做反弹运动。在调用 RenderScene( ) 函数前，可以建立一个连续改变物体坐标的循环，使正方形在窗口内移动，得到动画效果。

AUX 库提供了一个易于设置简单动画的函数 auxIdleFunc( )，使 OpenGL 程序在空闲时调用一个函数，函数的定义如下：

```
void CALLBACK IdleFunction(void);
```

除了处理移动窗口，改变窗口大小等事件外，AUX 库都在不断地调用 IdleFunction 函数。程序 3-4 是在程序 3-3 基础上做修改得到的，其功能是使方块在窗口内边框间反弹。因为每次画正方形前都要清除窗口，要浪费大量的 CPU 时间，即使在最好的硬件设备上运行这个程序获得的动画效果也不会好。为了得到光滑的动画，需要采用新方法。

程序 3-4

```
// Bouncing square

#include <windows.h>     // Standard windows include
#include <gl\gl.h>       // OpenGL library
#include <gl\glaux.h>    // AUX library

// Initial square position and size
GLfloat x1 = 100.0f;
GLfloat y1 = 150.0f;
```

```
GLsizei rsize = 50;

// Step size in x and y directions
// (number of pixels to move each time)
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;

// Keep track of windows changing width and height
GLfloat windowWidth;
GLfloat windowHeight;

// Called by AUX library when the window has changed size
void CALLBACK ChangeSize(GLsizei w, GLsizei h)
     {
     // Prevent a divide by zero, when window is too short
     // (you cant make a window of zero width).
     if(h == 0)
          h = 1;

     // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

     // Reset the coordinate system before modifying
    glLoadIdentity();

     // Keep the square square, this time, save calculated
     // width and height for later use
     if (w <= h)
          {
          windowHeight = 250.0f*h/w;
          windowWidth = 250.0f;
          }
    else
          {
          windowWidth = 250.0f*w/h;
          windowHeight = 250.0f;
          }
      // Set the clipping volume
      glOrtho(0.0f, windowWidth, 0.0f, windowHeight, 1.0f, -1.0f);
      }
```

```
// Called by AUX library to update window
void CALLBACK RenderScene(void)
    {
    // Set background clearing color to blue
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Set drawing color to Red, and draw rectangle at
    // current position.
    glColor3f(1.0f, 0.0f, 0.0f);
    glRectf(x1, y1, x1+rsize, y1+rsize);

    glFlush();
    }

// Called by AUX library when idle (window not being
// resized or moved)
void CALLBACK IdleFunction(void)
    {
    // Reverse direction when you reach left or right edge
    if(x1 > windowWidth-rsize || x1 < 0)
        xstep = -xstep;

    // Reverse direction when you reach top or bottom edge
    if(y1 > windowHeight-rsize || y1 < 0)
        ystep = -ystep;

    // Check bounds.  This is incase the window is made
    // smaller and the rectangle is outside the new
    // clipping volume
    if(x1 > windowWidth-rsize)
        x1 = windowWidth-rsize-1;

    if(y1 > windowHeight-rsize)
        y1 = windowHeight-rsize-1;

    // Actually move the square
    x1 += xstep;
    y1 += ystep;
```

```
    // Redraw the scene with new coordinates
    RenderScene();
    }

// Main body of program
void main(void)
    {
    // AUX window setup and initialization
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,250,250);
    auxInitWindow("Simple 2D Animation");

    // Set function to call when window is resized
    auxReshapeFunc(ChangeSize);

    // Set function to call when program is idle
    auxIdleFunc(IdleFunction);

    // Start main loop
    auxMainLoop(RenderScene);
    }
```

①双缓冲区

对用于制作动画的图形软件来说，最重要的特征在于是否支持双缓冲区。如果它支持双缓冲区，就可以在脱离屏幕的缓冲区内执行绘图命令、绘制场景，再用交换命令瞬时把图形显示在屏幕上。

双缓冲区有两个用途：绘制复杂物体要花很多时间，用户不必看到每一步骤的细节。使用双缓冲区，可以在完成一个图像后再进行组合、显示。用户不必看不完整的图像。

第二个用途是做动画。应用程序在脱离屏幕的缓冲区绘制每一帧，在准备好后迅速交换到屏幕上。OpenGL 支持双缓冲区特性，并且在 AUX 库中有支持双缓冲窗口的函数。使用这些函数对程序 3-4 加以改动，只需改动两处就可以得到光滑的动画。其一，改变 main( )中初始化显示模式的语句，使之标志可以使用双缓冲区。

```
auxInitDisplayMode(AUX_DOUBLE|AUX_RGBA);
```

这样，使绘图工作在脱离屏幕缓冲区内进行。其二，在 RenderScene( )的最后加入 auxSwapBuffers( )函数，把这个缓冲区的内容交换到屏幕，就可以得到满意的动画效果。

②绘制一些三维图形

在本章的前面部分提到，程序 3-3 绘制的矩形是在三维空间中进行的，只是因为裁剪体积与观察视线垂直，才使观察者看到的是一个平面矩形。下面将利用 AUX 库绘制一些真正的三维物体。AUX 库提供很多绘制三维实体、线框图的函数。它们以“auxSolid”或“auxWire”开头，分别用于绘制三维物体的实体和线框图，后面加上几何图形的名称，例如：调用

auxWireTeapot(50.0f)函数，可以绘制一个直径为 50.0 的茶壶的线框图。程序 3-5 定义的裁剪体积的三个坐标方向均为-100 到+100，得到如图 3-6 所示的图形，而 auxSolidTeapot(50.0f)将得到一个实体茶壶的轮廓，但要看到有真实感的茶壶，还要用到明暗处理、光照等技术。有关内容请参见本书的第八章。

程序 3-5

```
// 3D shape drawing with the aux library

#include <windows.h>     // Standard windows header
#include <gl\gl.h>       // Opengl library
#include <gl\glaux.h>    // AUX library

// The window has changed size
void CALLBACK ChangeSize(GLsizei w, GLsizei h)
     {
     // Prevent a divide by zero
     if(h == 0)
          h = 1;

     // Set viewport to match window size
     glViewport(0, 0, w, h);

     // Reset the coordinate system
    glLoadIdentity();

     // Setup clipping volume
     if (w <= h)
          glOrtho (-100.0f, 100.0f, -100.0f, 100.0f*h/w, -100.0, 100.0);
    else
          glOrtho (-100.0f, 100*w/h, -100.0f, 100.0f, -100.0, 1.00);
     }

// Draw the teapot
void CALLBACK RenderScene(void)
     {
     // Set clear color to blue and clear the screen
     glClearColor(0.0f, 0.0f, 1.0f, 0.0f);
     glClear(GL_COLOR_BUFFER_BIT);

     // Set drawing color to red and draw teapot
     glColor3f(1.0f, 0.0f, 0.0f);
     auxWireTeapot(50.0f);
```

```
    glFlush();
    }

// Main program body
void main(void)
    {
    // AUX window setup
    auxInitDisplayMode(AUX_SINGLE | AUX_RGBA);
    auxInitPosition(100,100,250,250);
    auxInitWindow("3D Wire-Frame Teapot");

    // Set window resize function
    auxReshapeFunc(ChangeSize);

    // Start main loop
    auxMainLoop(RenderScene);
    }
```

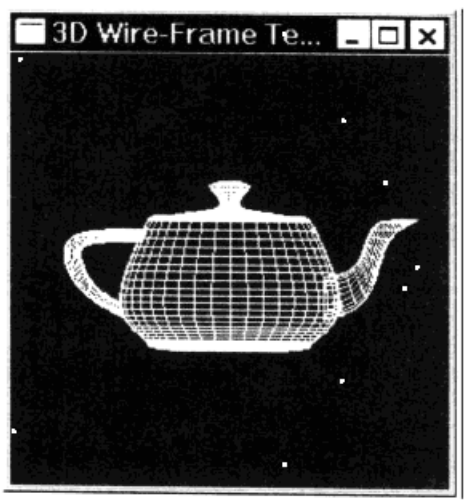


图 3-6　茶壶的线框图

3.6　相关函数

```
void auxInitDisplayMode(GLbitfield mask);
void auxInitPosition( GLint x, GLint y, GLsizei width, GLsizei height);
void auxInitWindow (GLbyte * titleString);
void CALLBACK auxReshapeFunc(void (*function) (GLsizei, GLsizei));
void CALLBACK auxKeyFunc(GLint key, void (*function) (void));
void CALLBACK auxMouseFunc(GLint button, GLint mode, void (*function) (AUX_EVENTREC*));
```

```
void auxSetOneColor(GLint index, GLfloat red, GLfloat green, GLfloat blue);
void auxWireSphere(GLdouble radius);
void auxSolidSphere(GLdouble radius);
void auxWireCube(GLdouble size);
void auxSolidCube(GLdouble size);
void auxWireBox(GLdouble width, GLdouble height, GLdouble depth);
void auxSolidBox(GLdouble width, GLdouble height, GLdouble depth);
void WireTorus(GLdouble innerRadius, GLdouble outerRadius);
void SolidTorus(GLdouble innerRadius, GLdouble outerRadius);
void auxWireCylinder(GLdouble radius, GLdouble height);
void auxSolidCylinder(GLdouble radius, GLdouble height);
void auxWireIcosahedron(GLdouble radius);
void auxSolidIcosahedron(GLdouble radius);
void auxWireOctahedron(GLdouble radius);
void auxSolidOctahedron(GLdouble radius);
void auxWireDodecahedron(GLdouble radius);
void auxSolidDodecahedron(GLdouble radius);
void auxWireCone(GLdouble radius, GLdouble height);
void auxSolidCone(GLdouble radius, GLdouble height);
void auxWireTeapot(GLdouble size);
void auxSolidTeapot(GLdouble size);
void CALLBACK auxIdleFunc(void *func);
void CALLBACK auxMainLoop(void (*displayFunc)(void));
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glFlush(void);
void glOrtho (GLdouble left, GLdouble right, GLdouble buttom, GLdouble top, GLdouble near,
GLdouble far);
void glViewport (GLint x, GLint y, GLsizei width, GLsizei height);
void glRect{dfis}(TYPE x1, TYPE y1, TYPE x2, TYPE y2);
```

第四章　Windows 环境下的 OpenGL

OpenGL 是一个纯粹的图形 API，由系统环境管理用户交互作用和屏幕/窗口。为简化这一依赖关系，每一环境没有对话框、没有用户界面，那么就应该学习如何在 Win32 应用程序中使用 OpenGL。

在本章开头，要创建一个能充分利用通常提供把 OpenGL“粘接”到自己窗口管理和用户接口函数的一些扩展函数。这个“胶水”是把 OpenGL 绘图命令与某一窗口相关的程序，同时也有必要提供设定缓冲区模式、颜色深度和其他绘图特性的函数。

在 Microsoft Windows 环境下，这些粘接程序是加入到 OpenGL 前缀为“wgl”的新函数和加入到 Windows NT 和 Windows95 GDI 的 Win32 函数。本章不再使用 AUX 库函数，而在 OpenGL 框架内使用这些新函数。

在第三章中，AUX 库是学习用 C 语言进行 OpenGL 编程的一个基本工具。讲述了如何绘制二维、三维图形以及如何确定坐标系统和取景透视体，但没有考虑 Windows 编程的细节。本章讲述在 Windows 环境下研究如何进行 OpenGL 编程。如果用户不满足于设计的 OpenGL 程序仅有一个窗口、没有菜单、没有打印功能、操作系统特性的 Windows 应用程序，讲述支持 OpenGL 图形的 Windows 窗口应该具备哪些特征，以及一个功能齐全的 OpenGL 窗口要处理哪些信息以及处理这些信息的方法。在用 C 语言创建 OpenGL 模型程序时，本章逐步引入这些概念，并为以后编写程序提供一个框架。

应当注意的是，尽管使用本书不必具备三维图形学和 C 语言编程知识，但至少假设读者已经有了一定的 Windows 编程知识。否则，就要先花些时间学习 Windows 编程，而不是学习 OpenGL 编程。

4.1　在 Windows 窗口下绘图

运行用 AUX 库函数编写的 OpenGL 程序时，仅能有一个窗口，OpenGL 只能在其中绘图，而 Windows 应用程序通常可以有多个窗口。事实上，对话框、控件、菜单是构成窗口的基础，通常仅有一个窗口的程序不可能是非常有用的程序。在 Windows 环境下编写程序前，有一些具体的问题需要考虑，例如如果有多个窗口，那么 OpenGL 又是如何确定在哪一窗口中执行绘图命令的呢？在回答这个问题前，先来看一看不用 OpenGL 是如何在窗口中绘图的。

4.1.1　GDI 设备描述表

如果不使用 OpenGL 在窗口中绘图，就要调用 Windows GDI (Graphical Device Interface) 函数。每个窗口都有一个接受图形输出的设备描述表，每个 GDI 函数把设备描述表作为标识函数对某一窗口起作用的参数。在程序中可以有多个设备描述表，但每个窗口仅能有一个。

程序 4-1（见与本书配套的软盘）的运行结果，与程序 3-2 绘制的图形完全相同。所不

同的是该程序完全由调用 Windows API 完成的。该程序是 Windows 编程风格的，有 WinMain 开始并保持信息流，WndProc 处理主窗口的信息。

下面仅给出绘制背景和正方形的程序段。

①首先要创建一个用于填充和绘画的蓝色、红色笔刷，其句柄全局定义为：

```
// Handles to GDI Brushes we will use for drawing
HBRUSH hBlueBrush,hRedBrush;
```

②在 WinMain 函数中创建笔刷，用 RGB 宏创建笔刷。

```
// Create a blue and red brush for drawing and filling Operations.
// Red, green, blue
hBlueBrush = CreateSolidBrush(RGB(   0, 0,     255));
hRedBrush = CreateSolidBrush(RGB(  255, 0,       0));
```

③在定义窗口风格后，在窗口类结构中用蓝笔刷设置背景。

```
wc.hbrBackground = hBlueBrush; // Use blue brush for background
```

④在创建窗口时，设置其位置和尺寸。

```
// Create the main application window
hWnd = CreateWindow(
                    lpszAppName,
                    lpszAppName,
                    WS_OVERLAPPEDWINDOW,
                    100, 100,                // Size and dimensions of window
                    250, 250,
                    NULL,
                    NULL,
                    hInstance,
                    NULL);
```

⑤最后，由 WM_PAINT 信息句柄在 WndProc 函数中完成绘图。

```
    case WM_PAINT:
        {
        PAINTSTRUCT ps;
        HBRUSH hOldBrush;

        // Start painting
        BeginPaint(hWnd,&ps);
        // Select and use the red brush
        hOldBrush = SelectObject(ps.hdc,hRedBrush);
        // Draw a rectangle filled with the currently
        // selected brush
        Rectangle(ps.hdc,100,100,150,150);
        // Deselect the brush
        SelectObject(ps.hdc,hOldBrush);
```

```
// End painting
EndPaint(hWnd,&ps);
}
break;
```

调用 BeginPaint 为在窗口中绘画做准备，并把 PAINTSTRUCT 结构的 hdc 成分赋给用于在该窗口绘图的设备描述表。这一设备描述表句柄作为所有 Windows GDI 函数标识对哪一窗口操作的第一参数，然后程序选定在绘图时用红笔刷，在(100, 100)，(150, 150)坐标处绘制一个填充矩形。其后不选用该笔刷，并由 EndPaint 函数清除绘图操作。

GDI 是 Windows 独有的，同时，设备描述表、窗口句柄或类似名词也不适用于其他环境。而 OpenGL 核心组成部分是在各种环境和硬件平台间可以完全移植的，所以采用设备描述表等概念设计的 OpenGL 程序也仅能在 Windows 环境下使用。

4.1.2 OpenGL 绘图描述表

为了实现 OpenGL 核心函数的可移植性，在执行任何一个 OpenGL 命令之前，在每个特定环境下，都必须有指定当前绘图窗口的方法。在 Windows 环境下，运行的 OpenGL 环境可以简单地概括为绘图描述表。正如设备描述表为 GDI 存贮绘图状态、命令的设置一样，绘图描述表存贮 OpenGL 的设置和命令。

在应用程序中可以有多个绘图描述表，使多个窗口有不同的绘图模式、透视体积等，但是为指定 OpenGL 命令针对的窗口，对每一个线程只能选定一个绘图描述表作为当前值。当一个绘图描述表是当前值时，它就与一个设备描述表和具体的窗口联系起来了。这样，系统也就知道了应该在哪一个窗口绘制 OpenGL 图形。图 4-1 说明 OpenGL 命令是如何通过绘图描述表与窗口相联系的。

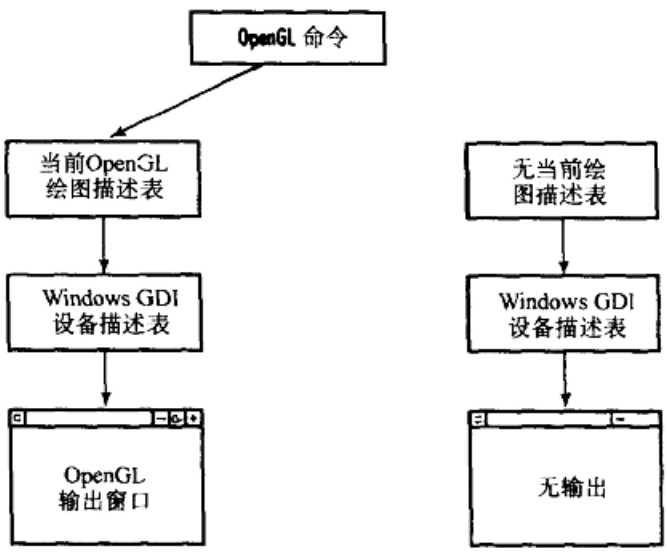


图 4-1 OpenGL 绘图描述表

OpenGL 支持多个线程同时在各自的窗口上绘图。这样，在多 CPU 的硬件系统上，可以大幅度提高绘图的效率。即使在仅有一个 CPU 的计算机上运行多线程应用程序，也可以在一个线程进行绘图操作，利用另一个线程处理用户界面，同样可以达到提高 OpenGL 效率的目

的。在与本书配套的软盘上给出一个多线程应用程序（程序 4-2），供读者参考、使用。该程序的执行结果如图 4-2 所示。

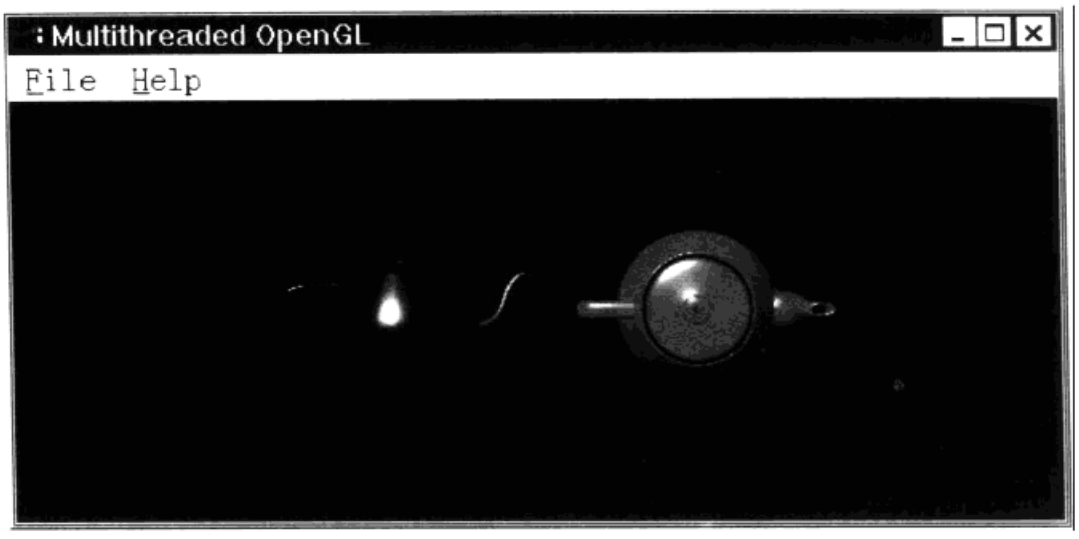


图 4-2 程序 4-2 的运行结果

①使用 wgl 函数

严格地说，绘图描述表不属于 OpenGL 的概念。实际上，这些新 wgl 函数是专门为使 Win32 API 支持 OpenGL 才加入其中的。与绘图描述表相关的三个函数是：

```
HGLRC wglCreateContext(HDC hdc);
BOOL wglDeleteContext(HGLRC hrc);
BOOL wglMakeCurrent(HDC hdc HGLRC hrc);
```

②创建并选择一个绘图描述表

新出现的数据类型 HGLRC 是绘图描述表的一个句柄。wglCreatContext 函数把这个句柄传递给 windows GDI 的设备描述表，并给 OpenGL 绘图描述表返回一个句柄。与 GDI 设备描述表相似，绘图描述表在使用后要进行删除操作，wglDeleteContext 函数就是完成这一功能的，其唯一的参数是要删除绘图描述表的句柄。

如果能为一个指定设备描述表创建绘图描述表，也就说明，OpenGL 可以在该设备描述表上绘图。值得注意的是，当用 wglMakeCurrent 函数把绘图描述表设为当前值时，所指定的设备描述表可以不必是开始创建绘图描述表的那个设备描述表，但是当一个绘图描述表是当前值时，所使用的设备描述表要与开始创建该绘图描述表的设备描述表有相同的特性。这些特性包括颜色深度、缓冲区定义等等，而这些特性又可以概括成像素格式的概念。

也就是说，把一个绘图描述表设置成当前值的设备描述表可以与创建它的设备描述表不同，但这两个设备描述表要有相同的像素格式。OpenGL 改变当前绘图描述表的方法有两个：即把其他绘图描述表变成为当前值；或者调用 wglMakeCurrent 函数，使该函数的参数改写为 NULL。

③用 OpenGL 绘图

如果读者在 GDI 编程方面没有深入研究，可能对设备描述表、绘图描述表的概念不易接受，但是在编写 OpenGL 程序时，使用中它们是非常简便易行的。在 16 位 Windows 环境编程时，需要检索设备描述表，在使用后，要释放该绘图描述表，因为 Windows 同时只能存贮 5

个设备描述表，而 32 后 Windows 没有这一限制。应该注意的是，在做动画时，一定要避免每次不断地调用 GetDC 函数，使绘图描述表变成当前值。如果把绘图描述表设成当前值后，不再改变它，可以节省时间；如果有个线程仅有一个窗口使用 OpenGL，也可以不必重复调用 wglMakeCurrent 函数，从而，节省大量的 CPU 时间。

在 Windows 环境下，程序创建和删除绘图描述表需要两个窗口信息：WM_CREATE 和 WM_DESTROY。WM_CREATE 信息用于创建绘图描述表，而 WM_DESTROY 删除绘图描述表。下面的程序段是一个 Windows 过程中创建、删除绘图描述表例子。

```
LRESULT CALLBACK WndProc(    HWND    hWnd,
                             UINT    message,
                             WPARAM  wParam,
                             LPARAM  lParam)
{
static HGLRC hRC;              // Permenant Rendering context
static HDC hDC;                // Private GDI Device context

switch (message)
     {
     // Window creation, setup for OpenGL
     case WM_CREATE:
          // Store the device context
          hDC = GetDC(hWnd);

          // Select the pixel format
          SetDCPixelFormat(hDC);

          hPalette = GetOpenGLFalette(hDC);

          // Create the rendering context and make it current
          hRC = wglCreateContext(hDC);
          wglMakeCurrent(hDC, hRC);
          SetupRC();

          break;

     // Window is being destroyed, cleanup
     case WM_DESTROY:
          // Deselect the current rendering context and delete it
          wglMakeCurrent(hDC,NULL);
          wglDeleteContext(hRC);
```

```
        if(hPalette != NULL)
            DeleteObject(hPalette);

        // Tell the application to terminate after the window
        // is gone.
        PostQuitMessage(0);
        break;
```

在程序中，由 WM_PAINT 信息处理窗口绘图，而不必使用 Beginpaint/EndPaint 命令。在 OpenGL 中，为预防 WM_PAINT 信息常量流传递到窗口，只需验证窗口绘图区域。下面是一个 WM_PAINT 句柄的框架：

```
case WM_PAINT;
        {
        // OpenGL drawing code or your render function called here
        RenderScence();

        ValidateRect(hWnd, NULL);
        }
break;
```

4.2 为 OpenGL 准备窗口

本节要用前面介绍的程序和绘图函数在 WM_PAINT 处理程序中写一个 Windows 程序。在创建绘图描述表前，还有下面两个步骤要准备。

①窗口风格

要用 OpenGL 在窗口中绘图，窗口的风格要设置成 WS_CHILDREN 和 WS_CLIPSBLINGS，而不能是 WS_PARENTDC。因为绘图描述表仅能在为其创建的、或者具有相同像素格式的窗口中适用。WS_CHILDREN 和 WS_CLIPSBLINGS 可以避免绘图函数改动子窗口。因为绘图描述表仅能与一个设备描述表和窗口相关联，所以不允许采用 WS_PARENTDC 窗口风格，从而使窗口继承上一级窗口的设备描述表。值得注意的是，若在程序中不事先指定窗口风格，也不能为窗口选定像素格式。

②像素格式

为了 OpenGL 能够在窗口中绘图，必须选定一个像素格式。与绘图描述表相似，像素格式也不是 OpenGL 本身的一部分，而是 Win32 API 为支持 OpenGL 功能而做的一个扩展。像素格式设置设备描述表的 OpenGL 属性，如颜色、缓冲区深度、窗口是否支持双缓冲区等。在使用设备描述表创建绘图描述表之前，一定要选择像素格式。下面是两个与设置像素格式有关的函数：

```
int ChoosePixelFormat(HDC hDC, PIXELFORMATDESCRIPTOR *ppfd);
BOOL SetPixelFormat(HDC hDC, int iPixelFormat, PIXELFORMATDESCRIPTOR *ppfd);
```

设置像素格式有三个步骤：首先，按照所需窗口的特性设置 PIXELFORMATDESCRIPTOR 结

构；把这一数据结构传给 ChoosePixelFormat 函数；该函数返回指定设备描述表可用像素格式的整型指数，这个指数传递给 SetPixelFormat 函数。上述步骤执行的顺序如下：

```
PIXELFORMATDESCRIPTOR pixelFormat;
int nFormatIndex;
HDC hDC;

// initialize pixelFormat structure
--
nFormatIndex= ChoosePixelFormat(HDC hDC, &pixelFormat);
BOOL SetPixelFormat(HDC hDC, nPixelFormat, &pixelFormat);
```

ChoosePixelFormat 函数把所支持的像素格式与 PIXELFORMATDESCRIPTOR 结构所需的信息匹配起来。如果用户需要屏幕显示 16M 颜色，而硬件仅能同时显示 256 色，则返回的像素格式是与之尽可能相近的一个近似值，如 256 色，然后把指数传递给 SetPixelFormat 函数。

下面给出建立 PIXELFORMATDESCRIPTOR 结构，为设备描述表设置像素格式的一个程序段。

```
void SetDCPixelFormat(HDC hDC)
    {
    int nPixelFormat;

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),  // Size of this structure
        1,                              // Version of this structure
        PFD_DRAW_TO_WINDOW |                    // Draw to Window (not to bitmap)
        PFD_SUPPORT_OPENGL,                        // Support OpenGL calls in window
        PFD_TYPE_RGBA,                          // RGBA Color mode
        24,                                     // Want 24bit color
        0,0,0,0,0,0,                            // Not used to select mode
        0,0,                                    // Not used to select mode
        0,0,0,0,0,                              // Not used to select mode
        16,                                     // Size of depth buffer
        0,                                      // Not used to select mode
        0,                                      // Not used to select mode
        PFD_MAIN_PLANE,                         // Draw in main plane
        0,                                      // Not used to select mode
        0,0,0 };                                // Not used to select mode
```

程序中并没有使用 PIXELFORMATDESCRIPTOR 结构的所有成分，表 4-1 给出所设定的成分，其余均可置为 0。

表 4-1 设置像素格式所需的 PIXELFORMATDESCRIPTOR 结构的成分

结构成分	解 释
nSize	结构的尺寸，由 sizeof(PIXELFORMATDESCRIPTOR)赋值
nVersion	数据结构的版本，设为 1
dwFlags	像素缓冲区的标识，设置为 PFD_DRAW_TO_WINDOW| PFD_SUPPORT_OPENGL|PFD_DOUBLEBUFFER
iPixelType	像素数据类型，对于 RGBA 模式，设置为 PFD_TYPE_RGBA
cColorBits	颜色位平面的数目，在此为 24 位，若硬件不支持 24 位，选择所支持的最大颜色位平面的数目
cDepthBits	深度缓冲区的深度，对于最大精度，选择 32，通常可为 16
iLayerType	层的类型，对于 OpenGL 的 Windows 实现，只能为 PFD_MAIN_PLANE

4.3 Windows 环境下的 OpenGL 程序

从上面介绍的内容可以看出，在 Windows 环境下，可以不用 AUX 库也能够创建一个支持 OpenGL 的 Windows 窗口。程序 4-3 是采用 wgl 和 Win32 函数后，对第三章的程序 3-4 所做的修改。可以看出，不用 AUX 库函数，尽管 OpenGL 的绘图窗口有了很多新功能，但是所编写的程序明显加长了。RenderScene，ChangeSize，IdleFunction 等函数均未改变，在此不再重复。程序 4-3 可以得到与程序 3-4 相同的结果。程序 4-3 中 WinMain 创建窗口和服务信息，WndProc 函数处理各个信息。

程序 4-3

```
// Entry point of all Windows programs
int APIENTRY WinMain(   HINSTANCE      hInstance,
                        HINSTANCE      hPrevInstance,
                        LPSTR          lpCmdLine,
                        int            nCmdShow)
    {
    MSG            msg;      // Windows message structure
    WNDCLASS  wc;            // Windows class structure
    HWND      hWnd;          // Storeage for window handle

    // Register Window style
    wc.style            = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc      = (WNDPROC) WndProc;
    wc.cbClsExtra       = 0;
    wc.cbWndExtra       = 0;
    wc.hInstance        = hInstance;
    wc.hIcon            = NULL;
    wc.hCursor              = LoadCursor(NULL, IDC_ARROW);

    // No need for background brush for OpenGL window
    wc.hbrBackground    = NULL;
```

```
    wc.lpszMenuName          = NULL;
    wc.lpszClassName    = lpszAppName;

    // Register the window class
    if(RegisterClass(&wc) == 0)
         return FALSE;

    // Create the main application window
    hWnd = CreateWindow(
                   lpszAppName,
                   lpszAppName,

                   // OpenGL requires WS_CLIPCHILDREN and WS_CLIPSIBLINGS
                   WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,

                   // Window position and size
                   100, 100,
                   250, 250,
                   NULL,
                   NULL,
                   hInstance,
                   NULL);

    // If window was not created, quit
    if(hWnd == NULL)
         return FALSE;

    // Display the window
    ShowWindow(hWnd, SW_SHOW);
    UpdateWindow(hWnd);

    // Process application messages until the application closes
    while( GetMessage(&msg, NULL, 0, 0))
         {
         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
    return msg.wParam;
    }

// Window procedure, handles all messages for this program
```

```
LRESULT CALLBACK WndProc(     HWND      hWnd,
                              UINT message,
                              WPARAM    wParam,
                              LPARAM    lParam)
    {
    static HGLRC hRC;         // Permenant Rendering context
    static HDC hDC;               // Private GDI Device context

    switch (message)
        {
        // Window creation, setup for OpenGL
        case WM_CREATE:
            // Store the device context
            hDC = GetDC(hWnd);

            // Select the pixel format
            SetDCPixelFormat(hDC);

            // Create the rendering context and make it current
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);

            // Create the palette
            hPalette = GetOpenGLPalette(hDC);

            // Create a timer that fires every millisecond
            SetTimer(hWnd, 101, 1, NULL);
            break;

        // Window is being destroyed, cleanup
        case WM_DESTROY:
            // Kill the timer that we created
            KillTimer(hWnd, 101);

            // Deselect the current rendering context and delete it
            wglMakeCurrent(hDC, NULL);
            wglDeleteContext(hRC);

            // Delete the palette
            if(hPalette != NULL)
                DeleteObject(hPalette);
```

```
        // Tell the application to terminate after the window
        // is gone.
        PostQuitMessage(0);
        break;

    // Window is resized.
    case WM_SIZE:
        // Call our function which modifies the clipping
        // volume and viewport
        ChangeSize(LOWORD(lParam), HIWORD(lParam));
        break;

    // Timer, moves and bounces the rectangle, simply calls
    // our previous OnIdle function, then invalidates the
    // window so it will be redrawn.
    case WM_TIMER:
        {
        IdleFunction();

        InvalidateRect(hWnd, NULL, FALSE);
        }
        break;

    // The painting function.  This message sent by Windows
    // whenever the screen needs updating.
    case WM_PAINT:
        {
        // Call OpenGL drawing code
        RenderScene();

        // Call function to swap the buffers
        SwapBuffers(hDC);

        // Validate the newly painted client area
        ValidateRect(hWnd, NULL);
        }
        break;

    // Windows is telling the application that it may modify
    // the system palette.  This message in essance asks the
    // application for a new palette.
    case WM_QUERYNEWPALETTE:
```

```
        // If the palette was created.
        if(hPalette)
            {
            int nRet;

            // Selects the palette into the current device context
            SelectPalette(hDC, hPalette, FALSE);

            // Map entries from the currently selected palette to
            // the system palette.  The return value is the number
            // of palette entries modified.
            nRet = RealizePalette(hDC);

            // Repaint, forces remap of palette in current window
            InvalidateRect(hWnd,NULL,FALSE);

            return nRet;
            }
        break;

    // This window may set the palette, even though it is not the
    // currently active window.
    case WM_PALETTECHANGED:
        // Don't do anything if the palette does not exist, or if
        // this is the window that changed the palette.
        if((hPalette != NULL) && ((HWND)wParam != hWnd))
            {
            // Select the palette into the device context
            SelectPalette(hDC,hPalette,FALSE);

            // Map entries to system palette
            RealizePalette(hDC);

            // Remap the current colors to the newly realized palette
            UpdateColors(hDC);
            return 0;
            }
        break;

    default:   // Passes it on if unproccessed
        return (DefWindowProc(hWnd, message, wParam, lParam));
    }
```

```
return (0L);
}
```

为与调用 AUX 库函数的程序 3-4 比较，并说明本章新出现的函数的具体使用方法，下面对程序 4-3 的各程序段做详细分析。

①缩放窗口

在第三章基于 AUX 库的程序 3-4 中，只要窗口尺寸发生变化，AUX 库就调用 ChangeSize 函数，而在本程序中需要捕捉 Windows 发出的 WM_SIZE 信息来调用 ChangeSize 函数，传递分别代表窗口新的宽度、高度值 lparam 的 LOWORD，HIWORD。

```
// Window is resized.
case WM_SIZE:
    // Call our function which modifies the clipping
    // volume and viewport
    ChangeSize(LOWORD(lParam), HIWORD(lParam));
    break;
```

②空闲时钟

AUX 在 CPU 空闲状态时，调用 IdleFunction。在此可以为窗口设置一个 Windows 定时器来模拟这一功能：

```
// Create a timer that fires every millisecond
SetTimer(hWnd,101,1,NULL);
```

在创建窗口时调用上面的函数，为窗口设置一个定时器。Windows 每间隔百万分之一秒向 OpenGL 窗口发出一个 WM_TIMER 信息。只要在应用程序信息序列中没有其他信息，就可以发出上述信息。当 WndProc 函数接收了 WM_TIMER 信息，就执行下面的程序：

```
case WM_TIMER:
    {
    IdleFunction();

    InvalidateRect(hWnd,NULL,FALSE);
    }
    break;
```

其中 IdleFunction 函数与第三章的内容相同，只是不调用 RenderScene( )，而是调用 InvalidRect 刷新图形，使 Windows 传递 WM_PAINT 信息。

③绘图

用于绘制 OpenGL 场景的程序段放在 WM_PAINT 处理程序中，该处理程序的作用是，调用 RenderScene 函数，交换缓冲区，校验窗口。

```
case WM_PAINT:
    {
    // Call OpenGL drawing code
    RenderScene();
```

```
// Call function to swap the buffers
SwapBuffers(hDC);

// Validate the newly painted client area
ValidateRect(hWnd,NULL);
}
break;
```

在程序中，发现了 Windows GDI 的新函数 SwapBuffer。SwapBuffer 与 auxSwapBuffers 函数的功能相同，其唯一参数是设备描述表。值得注意的是，该设备描述表的像素格式要设定 PFD_DOUBLEBUFFER 标识位，否则，函数调用失败。

至此，得到了一个在 Windows 环境下设计 OpenGL 应用程序的框架，可以使 OpenGL 窗口具备所有 Windows 窗口的属性。上面给出的程序段也可以在此基础之上作为有其他窗口、菜单等的成熟应用程序的一部分创建一个 OpenGL 窗口。

4.4 OpenGL 程序的调试

在本书的第三章和本章前面的内容中讲述了编写 OpenGL 程序的基本步骤。与一般意义上的程序设计相同，在 OpenGL 程序也可能出现各种问题，使程序的显示结果与希望的不一致，甚至有时屏幕上一片漆黑。要使程序能够通畅地运行，得到理想的效果，就要考虑可能出现的错误和一些没预料到的情况，OpenGL 提供了两种不同的对程序检验方法：

①查找错误。如果出现错误，可以停止执行并给出提示。

②对通用问题的简单解决方法，检查版本和 OpenGL 库的制造商。

OpenGL 内部有六组出错状态标识符，分别代表六类不同的错误。只要发生了其中的一个错误，相应的标识符被赋值。如果要知道这些标识符是否被赋值，就要调用 glGetError 函数。该函数的定义为：

```
GLenum glGetError(void);
```

glGetError 函数返回表 4-2 给出值中的一个。另外，GLU 库定义三个出错信息。需要注意的是，若同时有几个标识符被赋值，glGetError 仍然返回最后一个值。这个错误在下次调用 glGetError 时会被清除。再调用 glGetError 将返回其他出错标识或 GL_NO_ERROR。通常，要在循环中不停地检验出错标识符，直到返回值为 GL_NO_ERROR，才表示程序中没有 OpenGL 的错误。

表 4-2 glGetError 函数返回的错误代码

错 误 代 码	代 码 意 义
GL_NO_ERROR	没有出现错误
GL_INVALID_ENUM	枚举型变量的定义无效
GL_INVALID_VALUE	数值变量的值超出范围
GL_INVALID_OPERATION	进行了在当前状态下不允许的操作
GL_STACK_OVERFLOW	在堆栈上溢时执行了命令
GL_STACK_UNDERFLOW	在堆栈下溢时执行了命令
GL_OUT_OF_MEMORY	用于执行命令的内存不足

程序 4-4 是用于在循环中检查出错信息的程序。出错信息在如图 4-3 所示的对话框的控制件中出现。

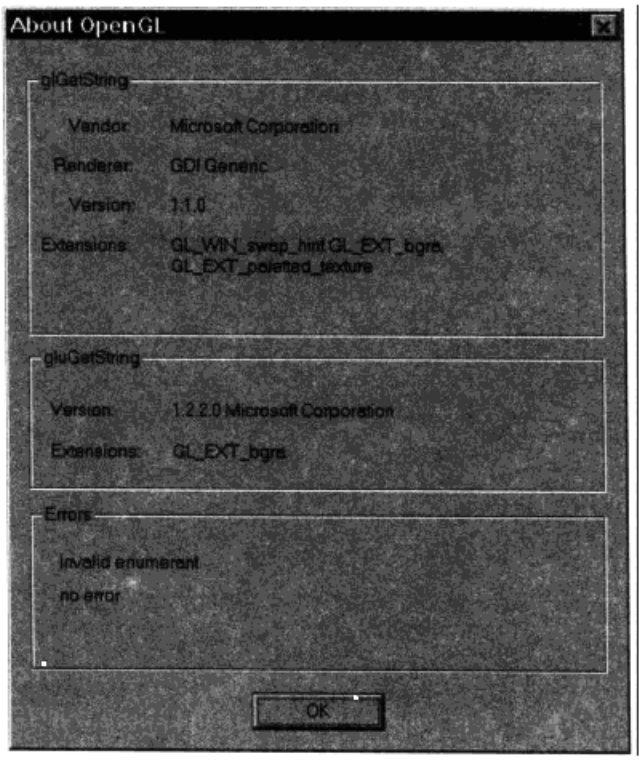


图 4-3　提示信息对话框

```
// Display any recent error messages
i = 0;
do {
     glError = glGetError();
     SetDlgItemText(hDlg, IDC_ERROR1+i, gluErrorString(glError));
     i++;
     }
while(i < 6 && glError != GL_NO_ERROR);
```

在 GLU 库中，用 gluErrorString 函数得到描述出错标识符的字符串，其定义为：

```
const GLubyte* gluErrorString(GLenum errorCode);
```

这个函数的参数是出错代码，返回一个描述该错误的静态字符串，如果错误代码为 GL_INVALID_ENUM，则返回“invalid enumerate”。

有时，错误是由于无效调用 OpenGL 函数或命令造成的。OpenGL 会忽略对该函数或命令的无效调用。这时 OpenGL 没有给出用户希望得到的结果，但是程序会继续执行下去。只有在出错代码为 GL_OUT_OF_MEMORY 或 GLU_OUT_OF_MEMORY 时是例外的，因为这些错误导致的 OpenGL 状态不确定，此时系统只好中断程序。

通过查询版本号和生产商信息，可以使用其提供的绘图引擎或 OpenGL 的扩展来加强程序的性能。GL 库和 GLU 库都可以返回上述信息。

对于 GL 库，可以调用 glGetString 函数。glGetString 函数的定义为：

```
const GLubyte* glGetString (GLenum name) ;
```

返回一个描述所需 GL 库信息的静态字符串，可使参数值为

```
const GLubyte* gluGetString (GLenum name) ;
```

下面给出程序 4-3 的一段程序，用于在 About 菜单中显示关于 GL、GLU 库版本和生产商信息，以及显示一个故意加入的错误。

```
// glGetString demo
SetDlgItemText(hDlg, IDC_OPENGL_VENDOR, glGetString(GL_VENDOR));
SetDlgItemText(hDlg, IDC_OPENGL_RENDERER, glGetString(GL_RENDERER));
SetDlgItemText(hDlg, IDC_OPENGL_VERSION, glGetString(GL_VERSION));
SetDlgItemText(hDlg, IDC_OPENGL_EXTENSIONS, glGetString(GL_EXTENSIONS));

// gluGetString demo
SetDlgItemText(hDlg, IDC_GLU_VERSION, gluGetString(GLU_VERSION));
SetDlgItemText(hDlg, IDC_GLU_EXTENSIONS, gluGetString(GLU_EXTENSIONS));
```

4.5 OpenGL 的扩展

OpenGL 的全部内容包括核心函数以及已经成为标准的组成部分，如本章介绍的 wgl 和 Win32 函数，还包括不断向其中加入的一些扩展功能。可以通过 GL_EXTENSIONS 和 GLU_EXTENSIONS 标识符得到在当前 OpenGL 的实现中生产商对 OpenGL 的扩展和一些流行但尚未成为标准的扩展功能。在编写程序时，充分利用这些扩展功能，可以大大提高 OpenGL 的性能，但是在使用前必须要做一些测试工作。

用 glHint 可以得到这方面的提示。用 wglGetProcAddress 可以进一步得到 OpenGL 扩展功能的信息。

glHint 函数允许用户选择不同类型操作的质量、速度的某种偏好。函数的定义如下：

```
void glHint(GLenum target, GLenum mode);
```

target 参数允许用户定义要修改的属性（参见表 4-3），而 mode 参数选择偏好程度，即用户关心绘图速度还是质量，或者都不关心，相应的变量值分别为 GL_FASTEST, GL_NICE, GL_DONT_CARE。

值得注意的是，并非所有的 OpenGL 实现都支持 glHint 函数，有时会忽略这些请求。

表 4-3 target 参数的选择

参 数	解 释
GL_FOG_HINT	影响雾效果的计算精度
GL_LINE_SMOOTH_HINT	影响反走样线的质量
GL_PERSPECTIVE_CORRECTION_HINT	影响颜色、纹理插值的质量
GL_POINT_SMOOTH_HINT	影响反走样点的质量
GL_POLYGON_SMOOTH_HINT	影响反走样多边形的质量

4.6 相关函数

```
int ChoosePixelFormat (HDC hdc, CONST PIXELFORATDESCRIPTOR *ppfd);
```

```
int DescribePixelFormat (HDC hdc, int iPixelFormat, UINT nBytes, LPPIXELFORMATDESCRIPTOR
ppfd);
int GetPixelFormat (HDC hdc);
Bool SetPixelFormat (HDC hdc, int iPixelFormat, CONST PIXELFORMATDESCRIPTOR *ppfd);
BOOL SwapBuffers (HDC hdc);
HGLRC wglCreateContext (HDC hdc);
BOOL wglDeleteContext( HGLRC hglrc);
HGLRC wglGetCurrentContext (void);
HDC wglGetCurrentDC (void);
PROC wglGetProcAddress(LPCSTR lpszproc);
BOOL wglMakeCurrent(HDC hdc, HGLRC hglrc);
BOOL wglShareLists(HGLRC hglrc1, HGLRC hglrc2);
BOOL wglUseFontBitmaps(HDC hdc, DWORD first, DWORD count, DWORD listBase);
BOOL wglUseFontOutlines (HDC hdc, DWORD first, DWORD count, DWORD listBase, FLOAT deviation,
FLOAT extrusion, int format, LPGLYPHMETRICSFLOAT lpgmf) ;
const GLubyte * gluErrorString (GLenum errorCode);
const GLubyte * gluErrorString (GLenum errorCode);
const GLubyte *glGetString(GLenum name);
GLenum glGetError(void);
```

第五章　在三维空间绘制几何要素

与搭积木用一些简单的形状可以构造出非常复杂和漂亮的结构一样，OpenGL 创建的物体和场景也是由较小、较简单的几何形状以不同方式组合而成的。OpenGL 绘制的最简单几何形状称为几何要素。OpenGL 的几何要素是一维、二维的物体，即点、线和多边形。本章将讲述如何在 OpenGL 程序中使用这些几何要素。

5.1　在三维空间画点

学习在计算机屏幕上绘图都是从像素开始的。像素是计算机屏幕上的最小单位。在彩色显示器上，像素可以有各种颜色。这是最简单的计算机图形学。首先学习如何在屏幕上画点，再赋上某一颜色，然后，以此为基础，可以画线、多边形、圆和其他几何形状等等。

但是用 OpenGL 在屏幕上绘图与上面提到的情况有着本质差别。使用 OpenGL 绘制图形时，用户不必关心屏幕坐标和像素，而是在取景体积中放置坐标。OpenGL 把在三维空间建立的点、线、多边形等所有一切都转换成计算机屏幕上的二维图像。

本章和下一章的内容都是讲述 OpenGL 及三维图形工具箱的基础概念的。下一章还要讲述如何把三维空间变换到计算机屏幕上的二维图像，以及如何控制所绘制的这些物体。只有掌握了几何要素以及坐标变换，用户才能深入理解任何一门三维图形学语言或 API。

1．设置一个三维画布

图 5-1 所示为一个简单取景体积，直角坐标 x, y, z 三个坐标轴的取值范围均为-100 到+100。可以把这个取景体积看成是 OpenGL 函数命令画图用的一块三维画布。可以调用 glOrtho( )函数创建这一体积。对窗口尺寸改变时调用的 ChangeSize( )函数的改动如下：

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat nRange = 100.0f;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
```

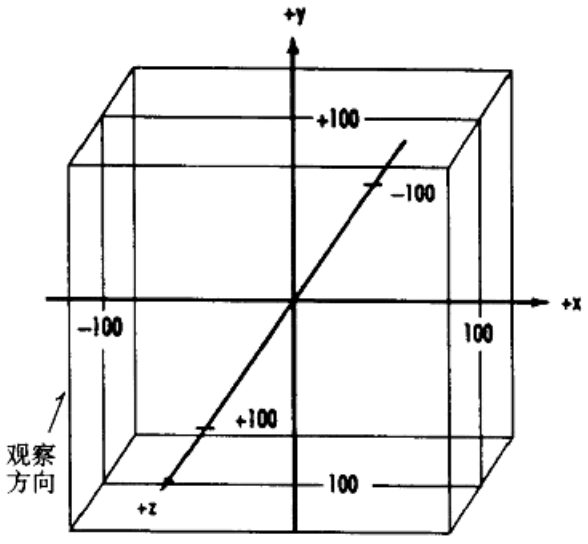


图 5-1　一个简单的取景体积

```
glLoadIdentity();

// Establish clipping volume (left, right, bottom, top, near, far)
if (w <= h)
      glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange, nRange);
else
      glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange, nRange);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
```

这一程序段与前面出现的相同名称的函数有些不同，其中出现了一些新函数，如 glMatrixMode，glLoadIdentity 等。这些函数将在第六章中详细介绍。

2．三维点：顶点

OpenGL 用 glVertex 函数在这块三维画布上指定一个绘图点。该函数是 OpenGL 程序中最常用的一个函数。glVertex 函数的参数数目从 2 到 4，数据类型从字节到双精度。

```
glVertex3f(50.0f, 50.0f, 0.0f);
```

在三维空间定义一个坐标为(50.0，50.0，0.0)的顶点。

```
glVertex2f(50.0f, 50.0f);
```

也代表相同的一点。该函数缺省的 z 值为 0.0。glVertex 函数的四个参数形式为 glVertex4。第 4 个坐标 w 是一个齐次坐标，用于放大、缩小图像，有关内容将在本书的第六章中详细介绍。

3．画图

顶点不仅是空间上的一个点，而是两条直线或曲线相交处的点，这才是几何要素的本质。几何要素是在屏幕上几个点组成的几何形状。OpenGL 有从空间一点到有任意边数的多边形十个不同形式的几何要素。可以用 glBegin 命令使 OpenGL 把一系列顶点理解成一个几

何要素。用 glEnd 命令结束对这个几何要素的描述。

4. 画点

```
glBegin(GL_POINTS);                    // Select points as the primitive
    glVertex3f(0.0f,0.0f,0.0f);        // Specify a point
    glVertex3f(50.0f,50.0f,50.0f);     // Specify another point
glEnd();                                   // Done drawing points
```

glBegin 的参数 GL_POINTS 表示 glVertex3f 函数给出的顶点是 OpenGL 的点，而上面程序段中的两个顶点就是两个指定的点。

从上面的用法可以看出，只要是相同类型的几何要素都可以在同一 glBegin/glEnd 间给出顶点。相比之下，若写成：

```
glBegin(GL_POINTS);
    glVertex3f(0.0f,0.0f,0.0f);        // Specify a point
glEnd();
glBegin(GL_POINTS);
    glVertex3f(50.0f,50.0f,50.0f);     // Specify another point
glEnd();
```

则会浪费很多时间。

程序 5-1 用三角函数 在三维空间绘制一些点，形成一个绕 z 轴的螺旋。SetupRC()函数把当前颜色设为绿色。

程序 5-1

```
void SetupRC()
    {
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

    // Set drawing color to green
    glColor3f(0.0f, 1.0f, 0.0f);
    }

// Called to draw scene
void RenderScene(void)
    {
    GLfloat x,y,z,angle; // Storeage for coordinates and angles

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
```

```
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Call only once for all remaining points
glBegin(GL_POINTS);

z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
    {
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);

    // Specify the point and move the Z value up a little
    glVertex3f(x, y, z);
    z += 0.5f;
    }

// Done drawing points
glEnd();

// Restore transformations
glPopMatrix();

// Flush drawing commands
glFlush();
}
```

在glBegin/glEnd之间的程序段计算x,y,z坐标，运行结果见图5-2。可以用箭头绕x,y轴旋转，这样，能更清楚地体现三维效果。

5．设置点的尺寸

缺省时，点的大小为一个像素。可以用glPointSize函数改变点的大小：

```
void glPointSize(GLfloat size);
```

该函数的唯一参数是所画点的直径近似值，但并不是所有值都能够做点的直径。用下面的函数可以得到点下的取值范围，以及这些值的最小间隔。

```
GLfloat sizes[2];    // Store supported point size range
GLfloat step;        // Store supported point size increments

// Get supported point size range and step size
glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

size数组的两个元素分别为glPointSize可用的最小值、最大值。Step是两个尺寸间的最小步长。OpenGL定义中只支持1.0直径，而Microsoft的OpenGL实现所支持的直径范围从0.5到10.0，间隔为0.125。超出这一取值范围就会出错，而选用与之最接近的最大值

图 5-2　程序 5-1 的显示结果

或最小值。

在下面的程序（程序 5-2）中采用了这些新函数，点的尺寸从最小值逐渐增大到最大值，运行结果如图 5-3 所示。

程序 5-2

```
// Called to draw scene
void RenderScene(void)
    {
    GLfloat x,y,z,angle; // Storeage for coordinates and angles
    GLfloat sizes[2];    // Store supported point size range
    GLfloat step;        // Store supported point size increments
    GLfloat curSize;     // Store current size

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Get supported point size range and step size
    glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
    glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

```
// Set the initial point size
curSize = sizes[0];

// Set beginning z coordinate
z = -50.0f;

// Loop around in a circle three times
for(angle = 0.0f; angle <= (2.0f*3.1415f)*3.0f; angle += 0.1f)
      {
      // Calculate x and y values on the circle
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);

      // Specify the point size before the primative is specified
      glPointSize(curSize);

      // Draw the point
      glBegin(GL_POINTS);
           glVertex3f(x, y, z);
      glEnd();

      // Bump up the z value and the point size
      z += 0.5f;
      curSize += step;
      }
```

图 5-3　改变点的尺寸

应该注意的是，调用 glPointSize 函数要放在 glBegin/glEnd 外，与之类似，还有一些函数也同样不能放在 glBegin/glEnd 之间。有关内容请参见本书作者编写的《精通 OpenGL》一书。

在程序 5-2 中，尺寸大的点在缺省情况下就是一个大方块。点的直径是所包含像素的近似值。当允许使用点光滑，即调用函数

```
glEnable(GL_POINT_SMOOTH);
```

OpenGL 就可以画出更接近小填充圆的、质量较高的点。还有另外的一些函数能影响到点、线的光滑，这些光滑方法都可以归纳为反走样技术。它可以光滑锯齿边缘，倒圆角。这部分内容详见本书第十三章。

5.2 在三维空间画线

1. GL_POINTS 参数

与上一节介绍的 GL_POINTS 相对应，glBegin 的另一个参数是 GL_LINES。它用于在 glBegin/glEnd 定义的两个顶点之间画一条直线。如果 glBegin/glEnd 间顶点数目为奇数，则忽略最后一个顶点。

```
glBegin(GL_LINES);                    // Select lines as the primitive
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(50.0f,50.0f,50.0f);
glEnd();
```

程序 5-3 是一个绘制直线的程序。运行的结果如图 5-4 所示。

程序 5-3

```
// Call only once for all remaining points
glBegin(GL_LINES);

z = 0.0f;
for(angle = 0.0f; angle <= GL_PI*3.0f; angle += 0.5f)
    {
    // Top half of the circle
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    glVertex3f(x, y, z);

    // Bottom half of the circle
    x = 50.0f*sin(angle+3.1415f);
    y = 50.0f*cos(angle+3.1415f);
    glVertex3f(x, y, z);
    }

// Done drawing points
glEnd();
```

图 5-4　用 GL_LINES 参数画直线

2. GL_LINE_STRIP 和 GL_LINE_LOOP 参数

这两个参数允许在所给出顶点间连线。GL_LINE_STRIP 参数 绘制连续线段。下面的程序段用三个顶点，绘制两条直线，如图 5-5 所示。

```
glBegin(GL_LINE_STRIP);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(50.0f,50.0f,50.0f);
    glVertex3f(50.0f,100.0f,50.0f);
glEnd();
```

GL_LINE_LOOP 与 GL_LINE_STRIP 相似，只是把 glBegin/glEnd 之间的第一个顶点和最后一个顶点连接起来，形成一条闭曲线。说明 GL_LINE_LOOP 参数的示意图参见图 5-6。

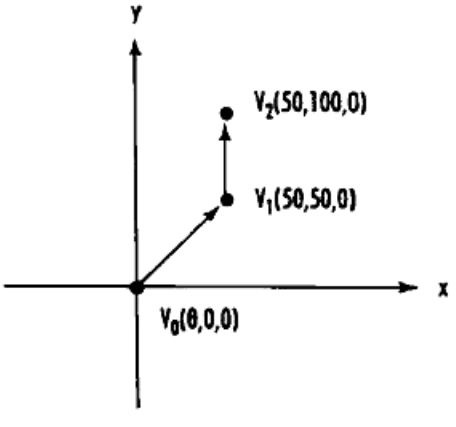


图 5-5　GL_LINE_STRIP 参数示意图

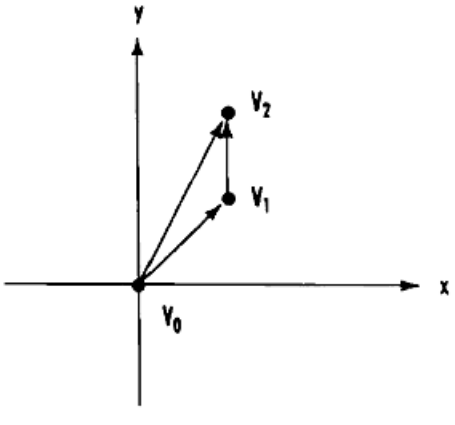


图 5-6　GL_LINE_LOOP 参数示意图

3. 用直线近似曲线

程序 5-4 把程序 5-1 的 GL_POINTS 用 GL_LINE_STRIP 代替，显示结果见图 5-7。可以看出，连接起来的线段是对曲线很好的近似。读者以后将发现在 OpenGL 程序中这种技巧无处不在。

程序 5-4

```
// Call only once for all remaining points
glBegin(GL_LINE_STRIP);

z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*3.1415f)*3.0f; angle += 0.1f)
    {
    x = 50.0f*sin(angle);
    y = 50.0f*cos(angle);
    // Specify the point and move the Z value up a little
    glVertex3f(x, y, z);
    z += 0.5f;
    }
// Done drawing points
glEnd();
```

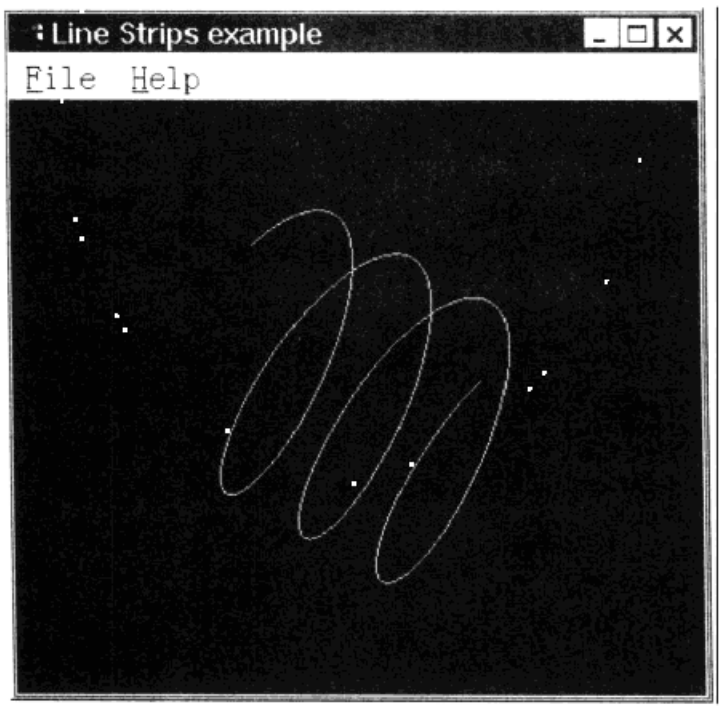


图 5-7　用线段近似曲线

4. 设置线宽

正如可以设置点的尺寸一样，在画线时，可以调用 glLineWidth 函数，指定以像素为单位的近似线宽。在设置线宽时，也会遇到相同的问题，即 OpenGL 并不支持所有线宽，下面的程序给出线宽的取值范围，以及不同线宽的最小间隔。

```
GLfloat size[2];              // Store supported line width range
GLfloat step;                 // Store supported line width increments

// Get supported line width range and step size
glGetFloatv(GL_LINE_WIDTH_RANGE, SIZE);
glGetfloatv(GL_LINE_WIDTH_GRANULARITY, &step);
```

size 数组的两个元素分别为 glLineWidth 函数可用的线宽的最小值、最大值，step 为不同宽度的最小间隔。OpenGL 定义的线宽为 1.0，而 Microsoft OpenGL 实现的线宽取值范围从 0.5 到 10.0，最小步长 0.125。

程序 5-5 为使用 glLineWidth 函数的一个例子。该程序绘制十条不同宽度的直线，由下至上每条线的宽度相差 1 个单位。如图 5-8 所示。

程序 5-5

```
// Called to draw scene
void RenderScene(void)
    {
    GLfloat y;                      // Storeage for varying Y coordinate
    GLfloat fSizes[2];              // Line width range metrics
    GLfloat fCurrSize;              // Save current size

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Get line size metrics and save the smallest value
    glGetFloatv(GL_LINE_WIDTH_RANGE,fSizes);
    fCurrSize = fSizes[0];

    // Step up Y axis 20 units at a time
    for(y = -90.0f; y < 90.0f; y += 20.0f)
        {
        // Set the line width
        glLineWidth(fCurrSize);

        // Draw the line
        glBegin(GL_LINES);
            glVertex2f(-80.0f, y);
            glVertex2f(80.0f, y);
```

```
                glEnd();

        // Increase the line width
        fCurrSize += 1.0f;
        }

    // Restore transformations
    glPopMatrix();

    // Flush drawing commands
    glFlush();
    }
```

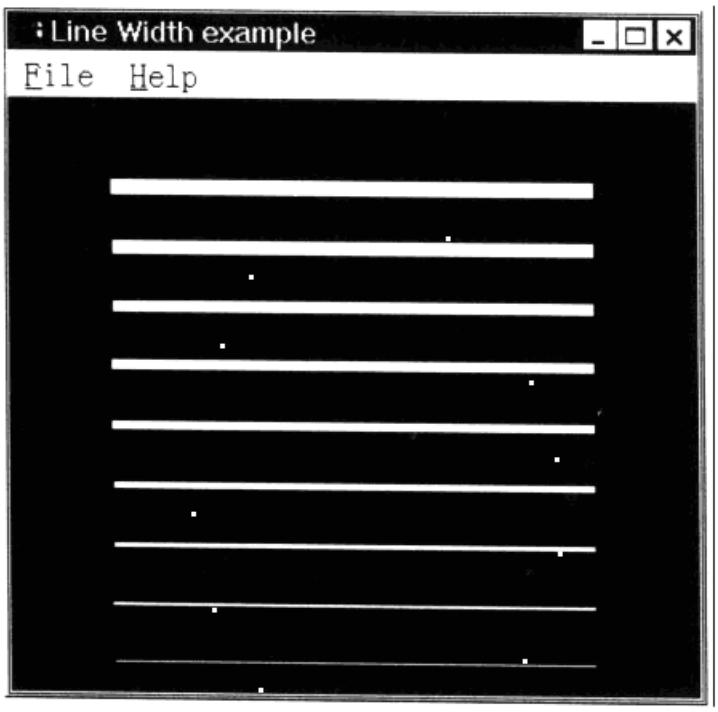


图 5-8 使用 glLineWidth 函数

除改变线宽外，也可以画点划线或虚线，称为线形，在使用这一功能前要调用：

```
glEnable(GL_LINE_STIPPLE);
```

然后才能用 glLineStipple 建立所需的线型。

```
void glLineStipple(GLint factor, GLushort pattern);
```

pattern 的参数是一个定义线型的 16 位二进制数。每一位代表是否画线段(0 表示不画线段，1 为画线段)。缺省时，每一位对应一个像素。factor 参数是线型宽度的系数，例如把 factor 设为 5，可以使每一位代表 5 个像素。

值得注意的是，使用 16 位二进制参数 Pattern 是从最低位开始的，这样可以节省宝贵的 CPU 时间。

程序 5-6 使用的线型参数为(01010101010101)，是由下至上绘制的条线，每线条的放大系数顺次加 1，如图 5-9 所示。

程序 5-6

```
// Called to draw scene
void RenderScene(void)
    {
    GLfloat y;                          // Storeage for varying Y coordinate
    GLint factor = 1;                   // Stippling factor
    GLushort pattern = 0x5555;          // Stipple pattern


    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Enable Stippling
    glEnable(GL_LINE_STIPPLE);

    // Step up Y axis 20 units at a time
    for(y = -90.0f; y < 90.0f; y += 20.0f)
        {
        // Reset the repeat factor and pattern
        glLineStipple(factor,pattern);

        // Draw the line
        glBegin(GL_LINES);
            glVertex2f(-80.0f, y);
            glVertex2f(80.0f, y);
        glEnd();

        factor++;
        }

    // Restore transformations
    glPopMatrix();

    // Flush drawing commands
    glFlush();
    }
```

图 5-9　程序 5-6 的显示结果

5.3　在三维空间画三角形

前面介绍了如何画点，画线。有了这些知识，可以绘制任意复杂的几何形状，但是应该看到，仅利用上述两个几何要素绘制的三维物体，是该物体的线框图，而不是实体，也不能在其中填充颜色。要绘制三维实体，需要用到多边形，它是 OpenGL 构成所有三维实体的基础。

1．三角形

三角形是最简单的多边形。任何多边形都可以分解成若干个三角形，并且多数图形加速卡也是针对绘制三角形而优化设计的，所以在编写 OpenGL 程序时，三角形通常是优先选用的几何要素。在 glBegin 函数中选用 GL_TRIANGLES 参数，可以用每三个顶点绘出一个三角形。下面的程序段绘制两个三角形，这两个三角形的示意图如图 5-10 所示。

```
glBegin(GL_TRIANGLE_FAN);
    glVertex2f(0.0f, 0.0f);          // V0
    glVertex2f(25.0f, 25.0f);        // V1
    glVertex2f(50.0f, 0.0f);         // V2

    glVertex2f(-50.0f, 0.0f);        // V3
    glVertex2f(-75.0f, 50.0f);       // V4
    glVertex2f(-25.0f, 0.0f);        // V5
glEnd();
```

值得注意的是，三角形会被填充上当前绘图颜色。如果不事先选定一个颜色，OpenGL 不会使用缺省颜色，从而，显示的结果不确定。

2. 旋转方向

图 5-10 所示的两个三角形分别按 V0, V1, V2, V0 和 V3, V4, V5, V3 的顺序连接而成，是顺时针方向的。定义顶点的顺序和方向称为旋转方向。缺省时，OpenGL 规定逆时针旋向的多边形是朝前的，顺时针是朝后的。这一点很重要，在编写 OpenGL 程序时，尤其是在场景中做了光照、剔除等处理的情况下，更加应该引起足够的重视。因为多边形的前、后面通常要赋予不同的特征，例如把朝后的多边形隐藏起来，或者具有不同的颜色、反射特性(有关内容请参见本书第八章)。在绘制场景时，保持旋转方向一致很重要。使用朝前多边形绘制实体做外表面，可以调用下面的函数改变 OpenGL 缺省设置，

```
glFrontFace (GL_CW);
```

使顺时针方向多边形为朝前的。同样，GL_CCW 参数 设定逆时针方向多边形是朝前的，这是 OpenGL 的缺省值。

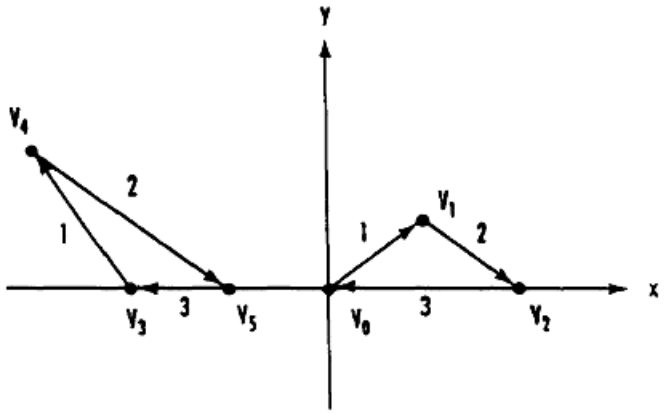


图 5-10 三角形的示意图

3. GL_TRIANGLE_STRIP

在许多情况下，要画几个相连的三角形。这时，使用 GL_STRIP 参数可以节省大量时间。图 5-11 为用 V0 到 V4 这 5 个顶点绘制 3 个带状排列的三角形的过程。可见，每个三角形的旋转方向相同，均为逆时针方向。

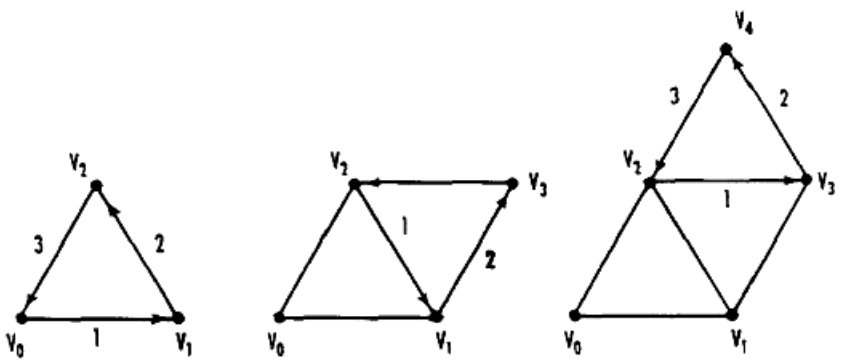


图 5-11 绘制带状排列三角形的过程

用带状排列的三角形的优势在于：

① 在给出第一个三角形的三个顶点后，只要再给一点，就可以画一个三角形。在绘制矢量三角形时，会节省大量时间；

② 在构成物体或表面时，要优于其他几何图像。

4. GL_TRIANGLE_FAN

GL_TRIANGLE_FAN 参数绘制绕一个中心点成扇形排列相互联接的三角形。图 5-12 绘出用 4 个顶点绘制四个三角形的示意图，所有的三角形旋向为顺时针方向。

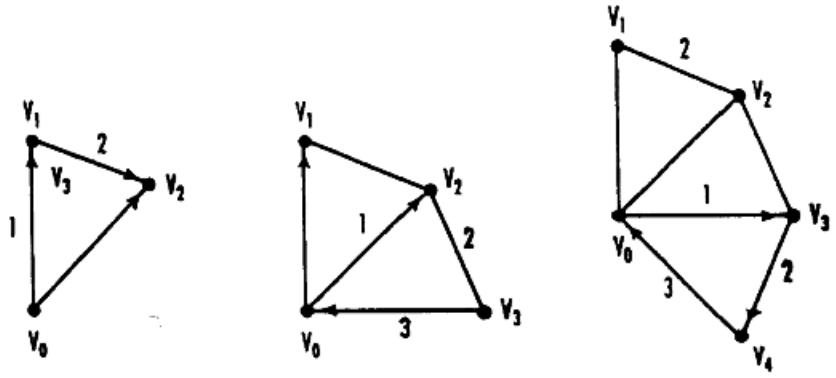


图 5-12　绘制扇形排列三角形的过程

5.4　创建三维实体

程序 5-7 用两组扇形排列的三角形分别构成锥形的表面和其底面。下面详细分析构成这个程序的主要部分。

程序 5-7

```
// This function does any needed initialization on the rendering
// context.
void SetupRC()
    {
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

    // Set drawing color to green
    glColor3f(0.0f, 1.0f, 0.0f);

    // Set color shading model to flat
    glShadeModel(GL_FLAT);

    // Clock wise wound polygons are front facing, this is reversed
    // because we are using triangle fans
    glFrontFace(GL_CW);
    }

// Called to draw scene
void RenderScene(void)
    {
```

```
GLfloat x,y,angle;  // Storage for coordinates and angles
int iPivot = 1;          // Used to flag alternating colors

// Clear the window and the depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Turn culling on if flag is set
if(bCull)
     glEnable(GL_CULL_FACE);
else
     glDisable(GL_CULL_FACE);

// Enable depth testing if flag is set
if(bDepth)
     glEnable(GL_DEPTH_TEST);
else
     glDisable(GL_DEPTH_TEST);

// Draw back side as a polygon only, if flag is set
if(bOutline)
     glPolygonMode(GL_BACK, GL_LINE);
else
     glPolygonMode(GL_BACK, GL_FILL);

// Save matrix state and do the rotation
glPushMatrix();
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Begin a triangle fan
glBegin(GL_TRIANGLE_FAN);

// Pinnacle of cone is shared vertex for fan, moved up Z axis
// to produce a cone instead of a circle
glVertex3f(0.0f, 0.0f, 75.0f);

// Loop around in a circle and specify even points along the circle
// as the vertices of the triangle fan
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
     {
     // Calculate x and y position of the next vertex
```

```
        x = 50.0f*sin(angle);
        y = 50.0f*cos(angle);

        // Alternate color between red and green
        if((iPivot %2) == 0)
             glColor3f(0.0f, 1.0f, 0.0f);
        else
             glColor3f(1.0f, 0.0f, 0.0f);

        // Increment pivot to change color next time
        iPivot++;

        // Specify the next vertex for the triangle fan
        glVertex2f(x, y);
        }

    // Done drawing fan for cone
    glEnd();

    // Begin a new triangle fan to cover the bottom
    glBegin(GL_TRIANGLE_FAN);

    // Center of fan is at the origin
    glVertex2f(0.0f, 0.0f);
    for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
         {
         // Calculate x and y position of the next vertex
         x = 50.0f*sin(angle);
         y = 50.0f*cos(angle);

         // Alternate color between red and green
         if((iPivot %2) == 0)
              glColor3f(0.0f, 1.0f, 0.0f);
         else
              glColor3f(1.0f, 0.0f, 0.0f);

         // Increment pivot to change color next time
         iPivot++;

         // Specify the next vertex for the triangle fan
         glVertex2f(x, y);
```

```
    }

// Done drawing the fan that covers the bottom
glEnd();

// Restore transformations
glPopMatrix();

// Flush drawing commands
glFlush();
}
```

图 5-13　程序 5-7 的显示

为突出这些三角形，将它们涂成红绿相间的颜色。这个程序演示了几个构造三维物体的注意事项。在 Effects 菜单中，可以打开关闭一些三维绘图特性演示所创建三维物体的特征。

1．设置多边形颜色

前面只讲述了如果设置一次当前颜色和绘制单个几何形状的方法。现在要画很多个多边形，程序中为便于观察，把它们赋予不同的颜色。为多边形的颜色赋值实际上是针对顶点，而不是针对多边形的。可以调用 OpenGL 设定明暗处理模式的函数 glShadeMode 完成上述赋值操作。OpenGL 的明暗处理模式有两种：

(1)用最后一个顶点的颜色填充多边形(GL_FLAT)；

(2)用每个顶点的颜色，使多边形上的颜色光滑过渡(GL_SMOOTH)。

有关内容请参见本书第七章。

2．消除隐藏面

在执行该程序时，可以按住键盘的一个方向键旋转圆锥。在旋转过程中，不难看出，因

为底部是在绘制完侧面后才画出的，无论圆锥在什么位置上，底部总是覆盖在侧面上的。窗口上显示的圆锥底面与侧面的相对位置是不正确的。同样的现象也出现在窗口上同时绘制多个三维物体，并且这些物体在观察者的视线方向能够遮挡时。可以说，上述现象在三维图形学是一个普遍现象，正确处理这个现象非常重要。

可以用消除隐藏面技术校正上述的怪现象。在 OpenGL 做这种处理就是进行深度检验。对应 Effect 菜单的 Depth Test 选项。具体的操作过程是，在屏幕上画一个像素时，记下该像素与观察者的距离(z 值)。如果在相同位置再绘制一个像素，比较这两个像素点的 z 值。若新像素的 z 值较大，即距观察者近，该点在已画出的像素前，遮住了已有的像素；反之，新像素被已有像素遮住。上述功能是由内部的深度缓冲区完成的，这部分内容请详见本书的第十二章。

设置深度检验功能，可以调用

```
glEnalbe(GL_DEPTH_TEST);
```

在程序 5-7 中，把 bDepth 置为 True 时，可以进行深度检验。

```
// Enable depth testing if flag is set
if(bDepth)
    glEnable(GL_DEPTH_TEST);
else
    glDisable(GL_DEPTH_TEST);
```

在选中 Effects 菜单的 Depth Test 选项时，给 bDepth 赋值 TRUE。深度缓冲区存贮像素与观察者间距离，用于确定一个像素是否会被另一像素遮住。在每次刷新屏幕时，都要把深度缓冲区复位。

```
// Clear the window and the depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

图 5-14 是做深度检验给出的显示结果。此时，程序对消隐面的处理是正确的。可见，进行深度检验是创建三维实体所必需的步骤。

图 5-14　深度检验的结果

3. 剔除

显而易见，如果绘图时不画出被遮住的物体会提高程序执行的效率，但是要付出的代价是绘制每个像素时都要与已画像素的 z 值相比较。有时，物体的某一部分是不必画的，如本例子中锥体的内部。通常 OpenGL 先画圆锥的背面，再画前面，然后比较 Z 缓冲区的值消除被遮住的部分。图 5-13，5-14 分别为做深度检验和不做深度检验的显示结果。做深度检验时，构成圆锥的红、绿三角形发生变化；不进行深度检验时，后面的锥面也是可见的。

前面提到 OpenGL 如何用旋向确定多边形是朝前或是朝后的，并且要使所定义物体外部的多边形旋向保持一致。这样，OpenGL 可以分辨出朝前、朝后的多边形。消除朝后的多边形可以大大节省绘制图像的时间。

去除朝前、朝后多边形的操作称为剔除。由下面的程序段控制这一功能。

```
    // Clock wise wound polygons are front facing, this is reversed
    // because we are using triangle fans
    glFrontFace(GL_CW);
    }

// Called to draw scene
void RenderScene(void)
    {
    GLfloat x,y,angle;  // Storage for coordinates and angles
    int iPivot = 1;          // Used to flag alternating colors

    // Clear the window and the depth buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Turn culling on if flag is set
    if(bCull)
        glEnable(GL_CULL_FACE);
    else
        glDisable(GL_CULL_FACE);
```

值得注意的是，由 GL_TRIANGLE_FAN 参数绘制的三角形均为顺时针方向，所以要把顺时针方向的多边形定义为朝前的。

图 5-15 所示为用剔除操作除去底部后的圆锥。因为没有遵守保持旋转方向一致性的规定，底部的三角形尽管是顺时针方向，但圆锥底部朝前面在圆锥内部。

在画第二个扇形排列三角形前可以用下面函数调用校正这一错误：

```
glFrontFace(GL_CCW);
```

4. 多边形模式

在缺省情况下，绘制的多边形要填充上颜色，用户也可以调用 glPolygonMode 函数把多边形画成轮廓线或几个点。可以把这一绘图模式用于朝前、朝后面的多边形。下面的程序段根据 bOutline 的值选择多边形模式。

```
    // Draw back side as a polygon only, if flag is set
```

```
if(bOutline)
    glPolygonMode(GL_BACK,GL_LINE);
else
    glPolygonMode(GL_BACK,GL_FILL);
```

图 5-16 中把朝后面的多边形画成轮廓线(此时，不允许做剔除操作)，圆锥底部变成了线框图，圆锥内壁也变成了由线框构成的三角形。

图 5-15　用剔除操作除去底部后的圆锥

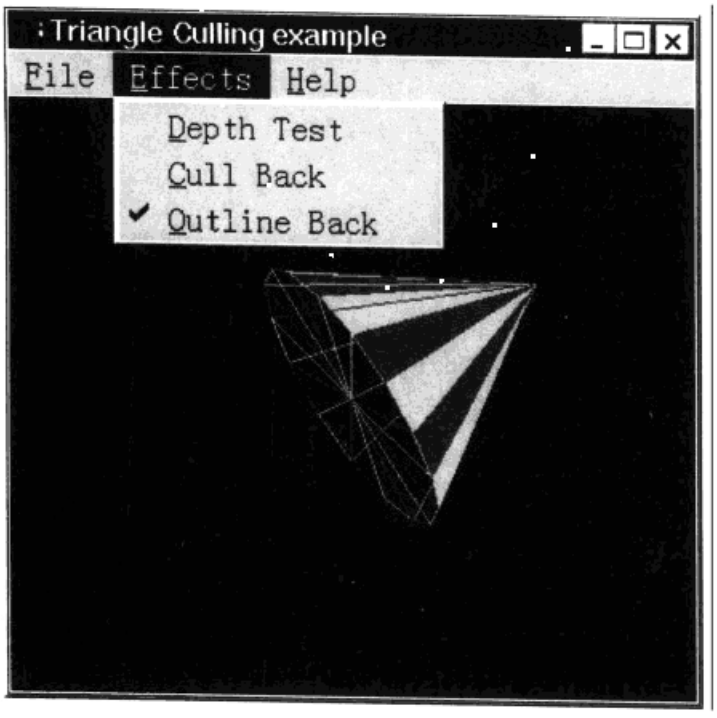


图 5-16　多边形轮廓线

5.5 其他几何要素

尽管绝大多数 OpenGL 硬件是专门为加速绘制三角形设计的，使三角形在构造物体时成为首选的几何要素，但也有为加速其他几何要素设计的硬件。使用通用几何要素在编程时更简单。除三角形外的几何要素有四边形、带状排列的四边形和通用多边形。

1．四边形

OpenGL 的 GL_QUADS 几何要素用于绘制一个四边形。该四边形由 glBegin/glEnd 之间四个顶点定义，其旋转方向为顺时针方向。

2．带状排列的四边形

与三角形的情况相似，GL_QUAD_STRIP 几何要素用于绘制带状排列的四边形，其旋向也为顺时针方向，如图 5-17 所示。

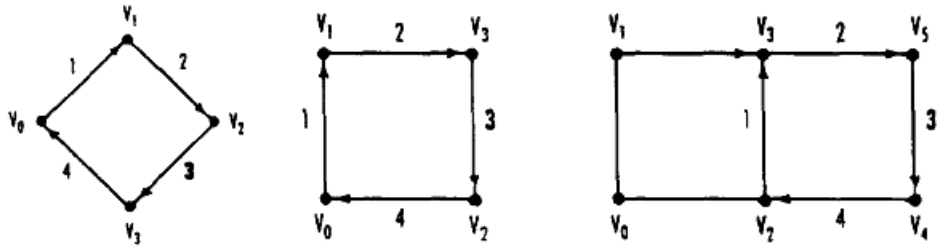


图 5-17　GL_QUADS 和 GL_QUAD_STRIP 参数的示意图

3．通用多边形

最后一个几何要素是 GL_POLYGON，用于绘制任意边数的多边形。其旋转方向也为顺时针方向。

4．多边形填充

有两种方法填充一个多边形：

①纹理映射，把一个位图映射到多边形表面(请参见本书第十章)；

②定义一个填充图案，多边形的填充图案是一个 32×32 位的单色位图。

在执行多边形填充操作前，要先调用

```
glEnable(GL_POLYGON_STIPPLE);
```

允许 OpenGL 使用该功能，然后再调用 glPolygonStipple 函数，

```
void glPolygonStipple(pBitmap);
```

其中 pBitmap 是指向包含位图数据数组的指针。该多边形就会被 pBitmap(GLubyte*)定义的图案填充。与使用线型数据不同的是，OpenGL 从最高位开始读数据。把填充图案存成数组的方式是由下至上按行存贮，每行由左至右，从最高位开始。因此，存贮的数组为 32×4 字节，见程序 5-8 所示，其第一行对应图案的最下面一行。

```
// Bitmap of camp fire
GLubyte fire[] = { 0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00,
                   0x00, 0x00, 0x00, 0x00,
```

```
    0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0xc0,
    0x00, 0x00, 0x01, 0xf0,
    0x00, 0x00, 0x07, 0xf0,
    0x0f, 0x00, 0x1f, 0xe0,
    0x1f, 0x80, 0x1f, 0xc0,
    0x0f, 0xc0, 0x3f, 0x80,
    0x07, 0xe0, 0x7e, 0x00,
    0x03, 0xf0, 0xff, 0x80,
    0x03, 0xf5, 0xff, 0xe0,
    0x07, 0xfd, 0xff, 0xf8,
    0x1f, 0xfc, 0xff, 0xe8,
    0xff, 0xe3, 0xbf, 0x70,
    0xde, 0x80, 0xb7, 0x00,
    0x71, 0x10, 0x4a, 0x80,
    0x03, 0x10, 0x4e, 0x40,
    0x02, 0x88, 0x8c, 0x20,
    0x05, 0x05, 0x04, 0x40,
    0x02, 0x82, 0x14, 0x40,
    0x02, 0x40, 0x10, 0x80,
    0x02, 0x64, 0x1a, 0x80,
    0x00, 0x92, 0x29, 0x00,
    0x00, 0xb0, 0x48, 0x00,
    0x00, 0xc8, 0x90, 0x00,
    0x00, 0x85, 0x10, 0x00,
    0x00, 0x03, 0x00, 0x00,
    0x00, 0x00, 0x10, 0x00 };
```

最后一行对应图案的最上面一行。把这个图案用于填充一个六边形，见图 5-18。有关程序段见程序 5-8。

程序 5-8

```
// This function does any needed initialization on the rendering context.
void SetupRC()
    {
    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    // Set drawing color to red
    glColor3f(1.0f, 0.0f, 0.0f);

    // Enable polygon stippling
    glEnable(GL_POLYGON_STIPPLE);
```

图 5-18 填充多边形

```
    // Specify a specific stipple pattern
    glPolygonStipple(fire);
    }
// Called to draw scene
void RenderScene(void)
    {
    // Clear the window
    glClear(GL_COLOR_BUFFER_BIT);

    // Save matrix state and do the rotation
    glPushMatrix();
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    // Begin the stop sign shape,
    // use a standard polygon for simplicity
    glBegin(GL_POLYGON);
        glVertex2f(-20.0f, 50.0f);
        glVertex2f(20.0f, 50.0f);
        glVertex2f(50.0f, 20.0f);
        glVertex2f(50.0f, -20.0f);
        glVertex2f(20.0f, -50.0f);
        glVertex2f(-20.0f, -50.0f);
        glVertex2f(-50.0f, -20.0f);
```

```
        glVertex2f(-50.0f, 20.0f);
    glEnd();
    // Restore transformations
    glPopMatrix();
    // Flush drawing commands
    glFlush();
    }
```

如果把多边形旋转一下，会发现填充图案并没有随多边形一起旋转，这是与纹理映射所不同的(请参见本书第十章的内容)。

5. 构造多边形的法则

OpenGL 为能采用快速算法绘制多边形，对所处理的多边形做了一定限制。如果用多边形构造一个复杂曲面，应该遵守如下二条法则:

①多边形应该是平面的，也就是说，多边形的顶点应该在同一平面上。多边形不能弯曲或扭转。由于在几何上，三点可以确定一个平面，所以使用三角形不会受到上述限制。

②多边形的边不能相交，即应该为凸多边形，如果贯穿多边形做一直线，该直线多次进入、离开四边形，则该多边形不符合规则。

尽管 OpenGL 只允许使用凸多边形，仍然可以创建出凹多边形。具体方法是把几个凸多边形排列在一起。如图 5-19 所示，可以用 6 个三角形构成一个凹多边形。在对这些多边形填充后，不能看到这些三角形的边缘。但是在轮廓线模式时，这些线可见。

OpenGL 为此提供了称为边缘标识的标识符。在定义顶点时，通过设定或清除该标识符，OpenGL 可以分辨出哪些线是轮廓线。glEdgeFlag 函数可以把边缘标识设置成 True 或 False。当标识符为 True 时，其后给出的顶点标志边界线的开始。见程序 5-9 所示。

程序 5-9

```
glBegin(GL_TRIANGLES);
    glEdgeFlag(bEdgeFlag);
    glVertex2f(-20.0f, 0.0f);
    glEdgeFlag(TRUE);
    glVertex2f(20.0f, 0.0f);
    glVertex2f(0.0f, 40.0f);

    glVertex2f(-20.0f,0.0f);
    glVertex2f(-60.0f,-20.0f);
    glEdgeFlag(bEdgeFlag);
    glVertex2f(-20.0f,-40.0f);
    glEdgeFlag(TRUE);

    glVertex2f(-20.0f,-40.0f);
    glVertex2f(0.0f, -80.0f);
    glEdgeFlag(bEdgeFlag);
    glVertex2f(20.0f, -40.0f);
```

```
        glEdgeFlag(TRUE);

        glVertex2f(20.0f, -40.0f);
        glVertex2f(60.0f, -20.0f);
        glEdgeFlag(bEdgeFlag);
        glVertex2f(20.0f, 0.0f);
        glEdgeFlag(TRUE);

        // Center square as two triangles
        glEdgeFlag(bEdgeFlag);
        glVertex2f(-20.0f, 0.0f);
        glVertex2f(-20.0f,-40.0f);
        glVertex2f(20.0f, 0.0f);

        glVertex2f(-20.0f,-40.0f);
        glVertex2f(20.0f, -40.0f);
        glVertex2f(20.0f, 0.0f);
        glEdgeFlag(TRUE);
    glEnd();
```

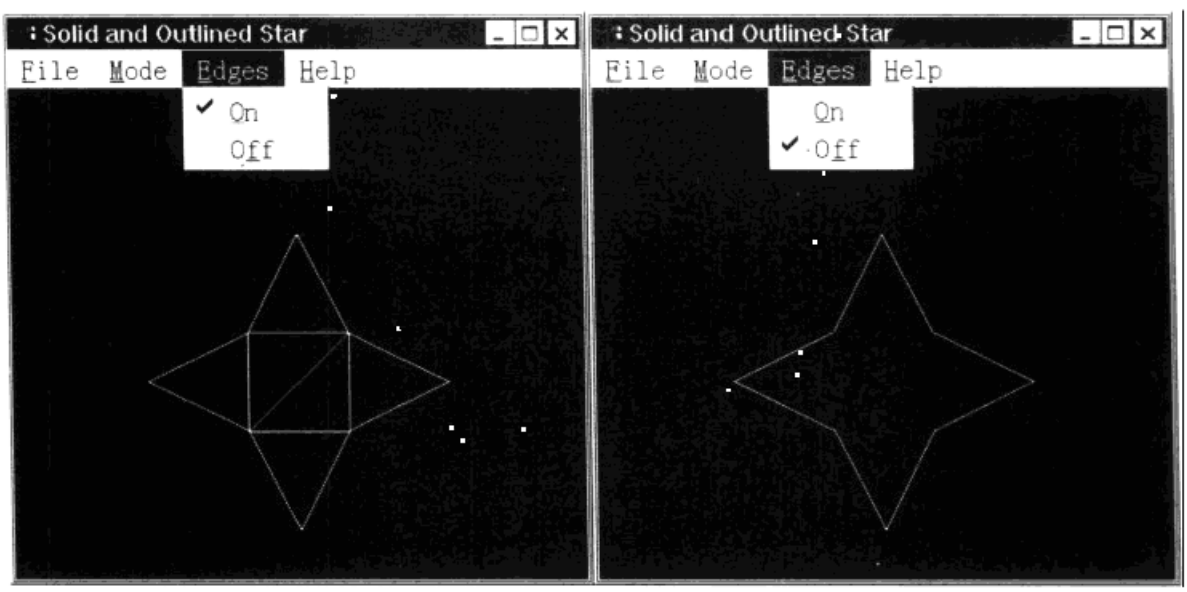


图 5-19　多边形边缘可见性

5.6　相关函数

```
void glBegin(GLenum mode);
void glEnd(void);
void glCullFace(GLenum mode);
```

```
void glEdgeFlag (GLboolean flag);
void glFrontFace (GLenum mode);
void glGetPolygonStipple (GLubyte *mask);
void glLineStipple (GLint factor, GLushort patterm);
void glLineWidth (GLfloat width);
void glPointSize (GLfloat Size)
void glPolygonMode (GLenum face, GLenum mode);
void glPolygonStipple (const GLubyte *mask);
void glVertex2{sifd}{v} (TYPE x, TYPE y);
void glVertex3{sifd}{v} (TYPE x, TYPE y, TYPE z);
void glVertex4{sifd}{v} (TYPE x, TYPE y, TYPE z, TYPE w);
```

第六章　坐标变换

在第五章中，介绍了在三维空间中画点、线和其他几何要素的方法。为把这些几何形状变成一个有序的场景，需要布置好它们之间的相互位置以及与观察者的位置。本章将讲述如何在坐标系统中控制这些物体。这是三维图形编程的一个关键步骤。

通过坐标变换可以移动、旋转物体，也可以对物体放大、缩小。在进行了坐标变换后，三维坐标的场景才能投影到二维屏幕上。坐标变换不是直接修改物体本身，而是通过改动坐标系统来实现的。从定义顶点到在屏幕上显示出图像的整个过程中，有三类变换：取景、建模和投影。从更一般意义上看，进行坐标变换实质是在改变描述坐标系统的矩阵。对物体的放大、缩小、旋转、平移等变换过程都可以写成矩阵相乘的数学表达式。本章介绍的 OpenGL 坐标变换原理并不是讲述烦琐的数学公式，在编写具体的应用程序时，用户也不必关心这些数学公式。OpenGL 在系统内部可以完成这些计算工作，用户只要掌握有关 OpenGL 坐标变换函数的使用方法就可以取得满意的效果。

6.1　对坐标变换的理解

1. 视觉坐标

视觉坐标并不是实际存在着的坐标，而是一个作为通用参考系而存在的虚拟固定坐标系统。视觉坐标不受变换影响，仅随观察点的选择而变化。本章讨论的所有变换都是相对于视觉坐标而言的。

图 6-1 所示为从两个取景点得到的视觉坐标系统。(a)为观察者在与屏幕垂直方向观察得到的视觉坐标；在(b)中，视觉坐标系统略有旋转，可以清楚地看到 z 轴。正 x, y 轴方向分别为向右、向上，正 z 轴指向观察者，负 z 轴指向屏幕。

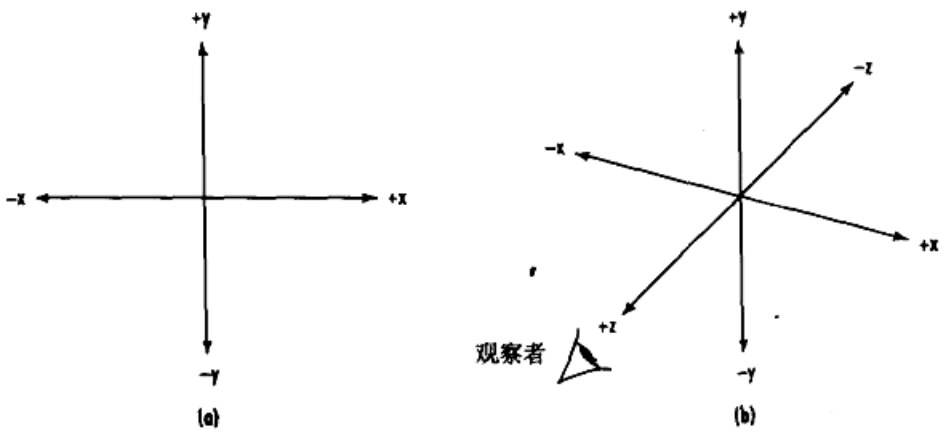


图 6-1　两个视觉坐标系统

使用 OpenGL 做三维图形所用的直角坐标，在不做任何变换时，与视觉坐标相同。如果

做坐标变换，则所有的变换都是相对于视觉坐标的，对当前坐标的改变。这就是如何相对于视觉坐标，在场景中旋转、移动物体的本质。

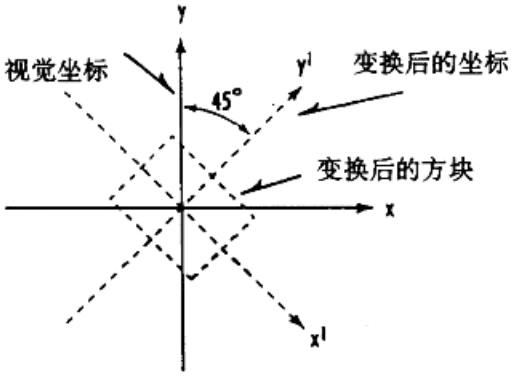


图 6-2　把坐标系统相对于视觉坐标旋转 45°

在修改当前坐标系统之前，用户可以存贮当前坐标系统。在变换、绘图后，再恢复原有状态，重新开始。这样，可以把物体布置在场景的不同位置和不同方位上。

2．取景变换

取景变换是对场景的第一次变换，用于确定观察场景的有利位置。缺省的观察点在(0, 0, 0)点，方向为 - z 轴方向。相对于视觉坐标移动这一点，可以得到一个有利于观察的位置，便于观察。当观察点在原点时，所绘制的 z>0 的坐标将在观察点的后面。

取景变换允许在任何位置、角度放置观察点。这一变换过程与在实际场景中放置照相机相似。因为取景变换相对于视觉坐标移动当前坐标系，其后的所有变换都要以此为基础，所以，取景变换要先于任何变换完成。

3．模型变换

模型变换用于旋转、移动、放大、缩小物体。变换后得到的场景或物体形状在很大程度上取决于做模型变换的顺序。图 6-3 所示说明这一顺序的重要性。(a)中先旋转再平移与(b)中先平移后旋转得到截然不同的效果。

4．模型取景变换

实际上，取景变换和模型变换得到的最终效果是完全相同的。所谓的差别只是由用户使用上的偏好所造成的，例如，向后移动物体与向前移动参考坐标系得到的最终效果是相同的。模型取景变换的意义在于可以把这些变换看成是模型变换，也可以看成是取景变换——其实并没有本质差别。

取景变换实质上也可以理解为在绘制物体前，观察者对这一虚拟物体所做的模型变换。如果在场景中绘制很多物体，就要不断地定义新变换。其后的变换都是基于初始变换的。

5．投影变换

投影变换作用于模型取景变换、定义取景体积并建立裁剪平面，或者说，投影变换定义如何把所完成的场景变换成屏幕上的图像。本章将讲述两类投影：正交投影和透视投影。

图 6-4 比较了正交投影和透视投影两者的差别。通常对不受观察者位置和距离影响的简单物体可以使用正交投影。当物体的尺寸相对于与观察者的距离非常小时，通常自然地选用正交视图。透视投影用于绘制多个物体在空间分开布置的场景，或者用于绘制改变观察点会使大物体发生变形的场合。

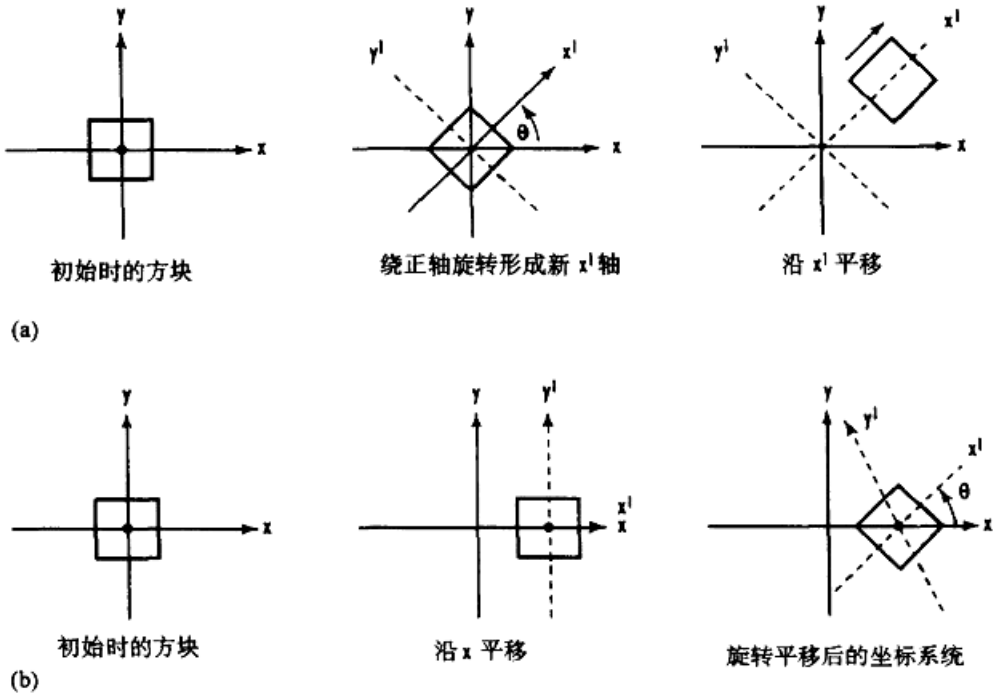


图 6-3　模型变换的顺序

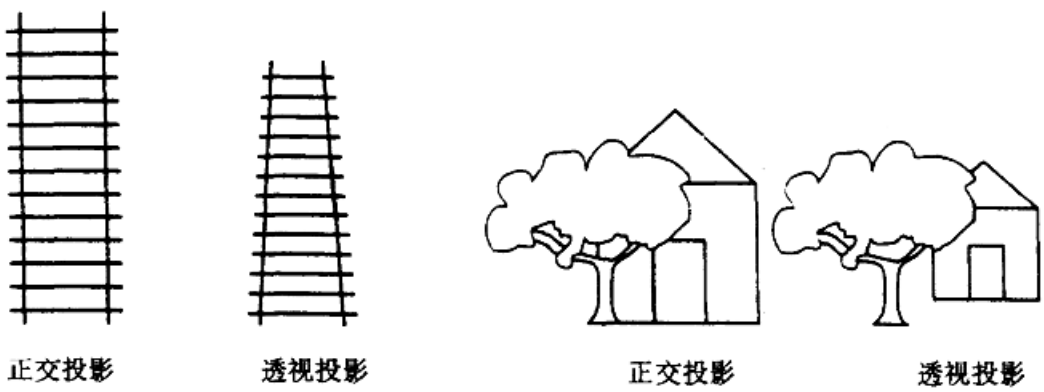


图 6-4　正交投影与透视投影

6. 视见区变换

在完成上述变换后，就要把所绘制场景的二维投影映射到屏幕上窗口的某一位置。这一物理窗口映射坐标的变换称为视见区变换。

下面讲述 OpenGL 是如何完成这些操作的，并介绍完成这些操作所要调用的函数。

6.2　OpenGL 的矩阵操作

简单地说，每一个变换都是用描述该变换的矩阵乘以包含顶点矩阵得到的结果。OpenGL 的所有变换都可以描述成两个或两个以上矩阵相乘。

只要修改 OpenGL 的模型取景矩阵和投影矩阵就可以完成上面提到的各个变换过程。OpenGL 提供了调用这些变换的高级函数。

图 6-5 给出的流程图描述如何把最初的顶点数据变成屏幕上的坐标。首先，顶点数据转换成 4×4 矩阵，前三个分量为 x, y, z 坐标，第四个分量为 w 坐标，(w 为一个缩放因子，x, y, z, w 构成三维齐次坐标)缺省值为 1.0。然后顶点坐标与模型取景矩阵相乘，变换成视觉坐标。再乘以投影矩阵变成裁剪坐标，消除取景体积之外的数据，除以 w 变成归一化的设备坐标。在变换过程中，w 值可能发生变化。值得注意的是，OpenGL 用高级函数完成这些功能，而无需用户进行矩阵乘法运算。

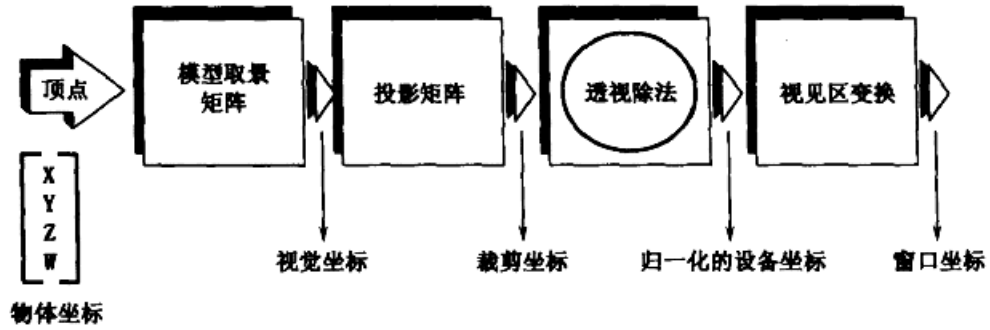


图 6-5 矩阵变换的过程

把三个分量坐标映射到二维屏幕上的视见区变换也可以表述成一个矩阵。这一矩阵不必直接设定或修改，而由 OpenGL 根据 glViewPort 函数的参数自动完成。

1. 模型取景矩阵

模型取景矩阵是表示放置物体位置和方向的变换坐标系统的 4×4 矩阵。顶点数据矩阵与模型取景矩阵相乘得到相对于视觉坐标的变换坐标。

①平移

OpenGL 执行平移操作的函数为 glTranslatef，其定义为：

```
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

其中，参数 x, y, z 分别为沿 x, y, z 轴三个方向平移的量。

```
// Translate up the y-axis 10 units
glTranslatef(0.0f, 10.0f, 0.0f);
```

沿 y 轴平移 10 个单位长度，如图 6-6 所示。

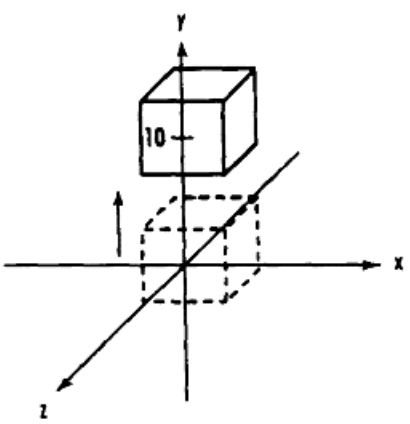


图 6-6 沿 y 轴平移物体

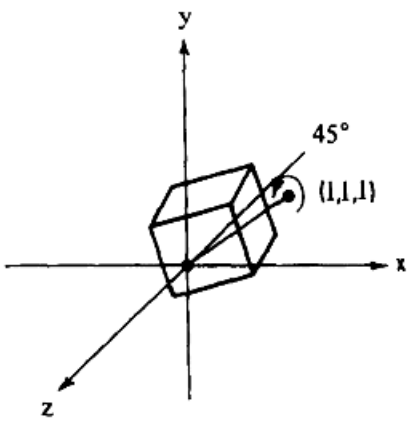


图 6-7 绕任意一轴旋转物体

②旋转

OpenGL 执行旋转操作的函数为：

```
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

物体绕着从(0,0,0)指向(x,y,z)矢量方向逆时针旋转 angle 角度。

```
// Perform the rotation transformation
glRotatef(45.0f,1.0f, 1.0f, 1.0f);
```

绕(1,1,1)矢量逆时针旋转 45°，如图 6-7 所示。

③缩放

缩放操作按照缩放系数指定的倍数分别改变沿三个坐标轴的尺寸。

```
void glScalef(GLfloat x, GLfloat y, GLfloat z);
```

x,y,z 三个系数不必相同，也就是说，三个坐标方向的缩放倍数可以不一致。

```
// Perform the Scaling transformation
glScalef(2.0f, 1.0f, 2.0f);
```

在 x,z 轴方向伸长一倍，y 方向保持不变,如图 6-8 所示。

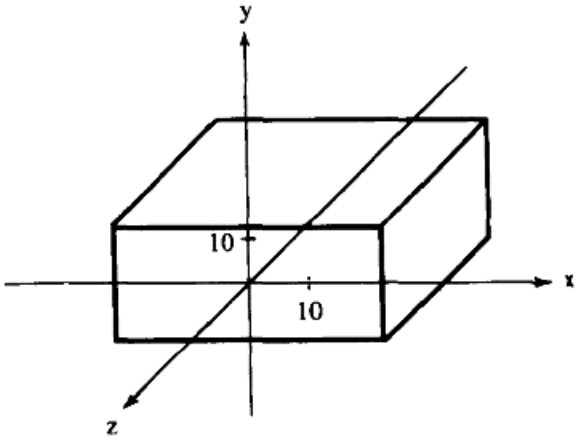


图 6-8　缩放物体

6.3　矩阵操作

1．单位矩阵

前面提到过，变换的过程实质上是两个或多个矩阵相乘得到的结果。矩阵运算特性决定了所调用变换函数对后面的操作也有影响。调用这些变换函数也就构造了一个矩阵，并与当前模型取景矩阵相乘，使所得到的矩阵为当前矩阵，因此，在程序开始时，一定首先要调用单位矩阵，把模型取景矩阵复位。

下面举例说明使用单位矩阵的重要性。如果要用

```
// Go 10 units up the y-axis
glTranslatef(0.0f, 10.f,0.0f);

// Draw the first sphere
auxSolidSphere(1.0f);
```

```
// Go 10 units out the x-axis
glTranslatef(10.0f, 0.f,0.0f);

// Draw the second sphere
auxSolidSphere(1.0f);
```

分别在 x，y 坐标轴上绘制两个物体，得到的两个球如图 6-9 所示，而不是想象的图 6-10。正确的绘图顺序应该是：

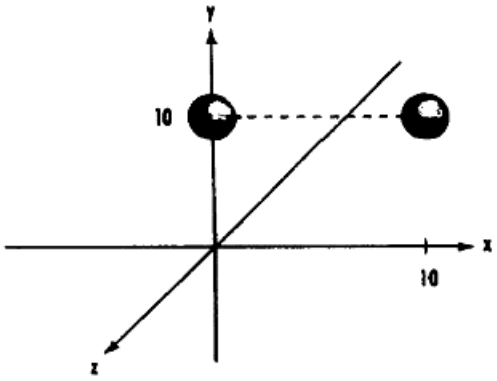


图 6-9　错误的程序结果

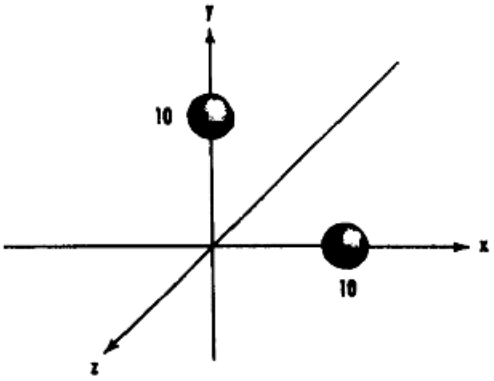


图 6-10　正确的程序结果

```
// Set current matrix to Modelviev and reset
glMatrixmode(GL_MODELVIEV);
glLoadIdentity();

// Go 10 units up the y-axis
glTranslatef(0.0f, 10.f,0.0f);

// Draw the first sphere
auxSolidSphere(1.0f);

// Reset Modelview Matrix
glLoadIdentity();

// Go 10 units out the x-axis
glTranslatef(10.0f, 0.f,0.0f);

// Draw the second sphere
auxSolidSphere(1.0f);
```

下面的程序段把模型取景矩阵置为单位阵。

```
glMatrixmode(GL_MODELVIEV);
glLoadIdentity();
```

第一行定义下面所有的操作都针对模型取景矩阵；第二行把当前矩阵置为单位阵。读入

单位阵的作用在于不对顶点做任何变换。

2. 矩阵堆栈

有时，要把当前矩阵变换的状态储存起来，在绘制出一些物体后再恢复原有的矩阵变换。OpenGL 为模型取景矩阵和投影矩阵的存取提供了矩阵堆栈。该矩阵堆栈与普通堆栈的工作原理相同。glPushMatrix，glPopMatrix 函数分别把变换矩阵推入、弹出矩阵堆栈。

可以用下列两个函数分别检索模型取景堆栈和投影堆栈深度的最大值。

```
glGet(GL_MAX_MODELVIEV_STACK_DEPTH);
glGet(GL_MAX_PROJECTION_STACK_DEPTH);
```

若向堆栈推入的矩阵数目超过了堆栈深度的最大值，则发生堆栈上溢，会出现 GL_STACK_OVERFLOW 错误；如果弹出空堆栈矩阵，堆栈下溢，得到 GL_STACK_UNDERFLOW 错误信息。堆栈的深度与具体的 OpenGL 实现有关。在 Microsoft 的 OpenGL 实现中，模型取景矩阵堆栈的最大深度为 32，投影矩阵堆栈深度为 2。

3. 一个原子核例子

现在让我们把上面讲述的内容综合用于一个实际例子（程序 6-1）。在这个例子中，用一个位于中心的球代表原子核，三个电子围绕这个球转动。在程序中，用一个定时器使电子每秒动四次。每次调用绘图函数后，给电子的角度一个增量。每个电子位于不同的平面。程序 6-1 给出这个例子的绘图函数，图 6-11 为显示结果。

程序 6-1

```
// Called to draw scene
void RenderScene(void)
    {
    // Angle of revolution around the nucleus
    static float fElecti = 0.0f;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Reset the modelview matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Translate the whole scene out and into view
    // This is the initial viewing transformation
    glTranslatef(0.0f, 0.0f, -100.0f);

    // Red Nucleus
    glRGB(255, 0, 0);
    auxSolidSphere(10.0f);

    // Yellow Electrons
    glRGB(255,255,0);
```

```
// First Electron Orbit
// Save viewing transformation
glPushMatrix();

// Rotate by angle of revolution
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);

// Translate out from origin to orbit distance
glTranslatef(90.0f, 0.0f, 0.0f);

// Draw the electron
auxSolidSphere(6.0f);

// Restore the viewing transformation
glPopMatrix();

// Second Electron Orbit
glPushMatrix();
glRotatef(45.0f, 0.0f, 0.0f, 1.0f);
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
glTranslatef(-70.0f, 0.0f, 0.0f);
auxSolidSphere(6.0f);
glPopMatrix();

// Third Electron Orbit
glPushMatrix();
glRotatef(360.0f-45.0f,0.0f, 0.0f, 1.0f);
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
glTranslatef(0.0f, 0.0f, 60.0f);
auxSolidSphere(6.0f);
glPopMatrix();

// Increment the angle of revolution
fElect1 += 10.0f;
if(fElect1 > 360.0f)
    fElect1 = 0.0f;

// Flush drawing commands
glFlush();
}
```

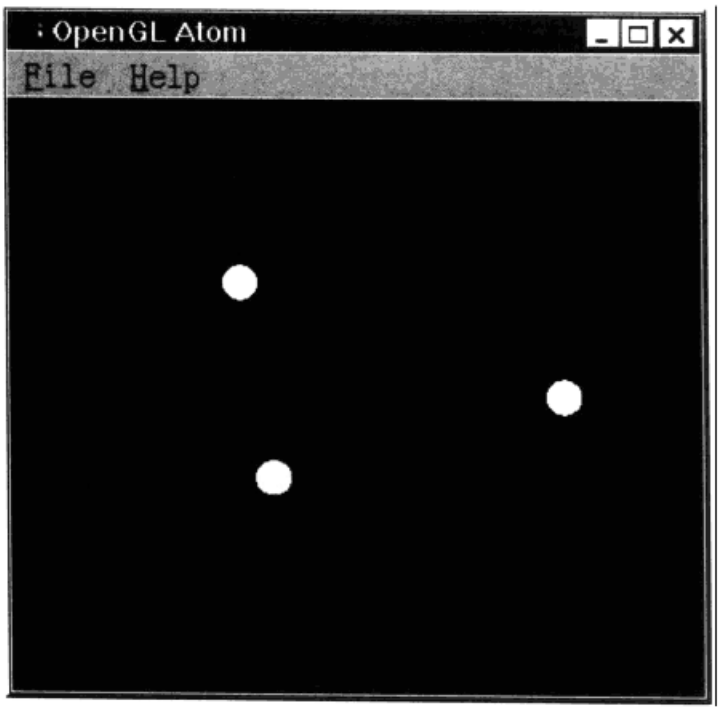


图 6-11　程序 6-1 的显示结果

下面详细分析程序 6-1 中是如何进行坐标变换的。

①使用 glPushMatrix 把当前变换矩阵推入堆栈，保存当前模型取景矩阵，

```
// First Electron
// Save viewing transformation
glPushMatrix();
```

②把坐标系统绕 y 轴旋转 fElect1 角度，

```
// Rotate by angle of revolution
glRotatef(fElect1, 0.0f, 1.0f, 0.0f);
```

③把刚刚旋转的坐标系统平移，并绘制出一个实心球，代表电子。

```
// Translate out from origin to orbit distance
glTranslatef(90.0f, 0.0f, 0.0f);
```

④把模型取景矩阵从堆栈上弹出，恢复到绘制开始时的模型取景矩阵。

```
// Draw the electron
auxSolidSphere(6.0f);

// Restore the viewing transformation
glPopMatrix();
```

绘制其余两个电子的过程与上面的相似。

6.4　使用投影

在前面所举的例子中可以看到，模型取景矩阵用于在取景体积内占据有利位置并把物体

放入其中，而投影矩阵用于指定取景体积的大小及形状。

前面介绍过如何使用 glOrtho 函数设定远近、左右、上下裁剪坐标来创建简单的平行取景体积。当把投影矩阵置为单位阵时，裁剪平面在各个方向上都从原点到+1。尽管投影矩阵并没有缩放和调整透视的功能，但是有实现这些操作的其他方法。

1. 正交投影

正交投影的每一侧都是正方形。各个方向宽度相同。正交投影得到的是平行的投影。如图 6-12 所示。

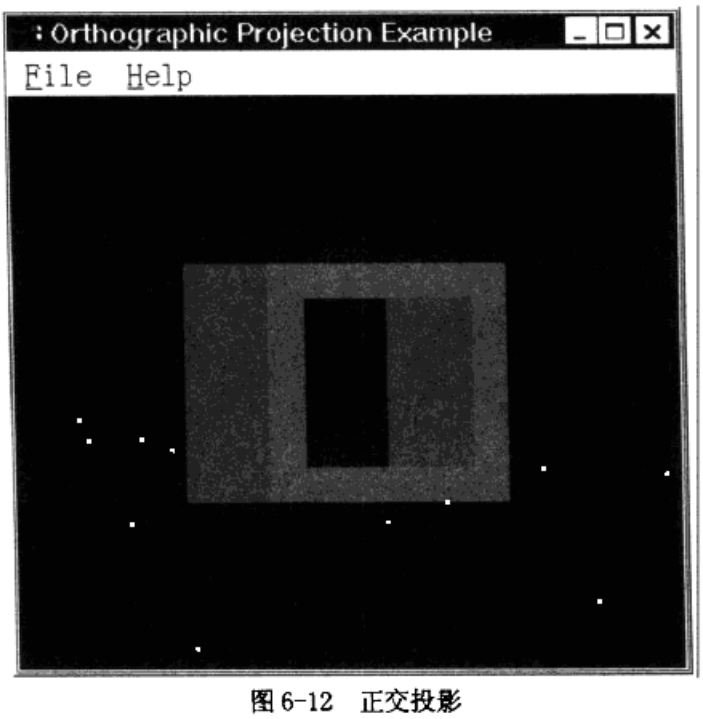


图 6-12 正交投影

2. 透视投影

透视投影的取景体积的前后宽度不一致，相同尺寸放置在取景体积的前面要比在后面小。透视投影的取景体积为一个平截头锥体，可以由 glFrustum 函数定义该锥体的形状，如图 6-13 所示，但是使用该函数设定投影取得所希望的效果并不直观。读者可以方便地使用 glu 库提供的 gluPerspective 函数：

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

gluPerspective 函数的参数 fovy 是垂直方向的视角，aspect 是高度与宽度的比值，zNear， zFar 分别为从近到远处裁剪平面的距离，如图 6-14 所示。

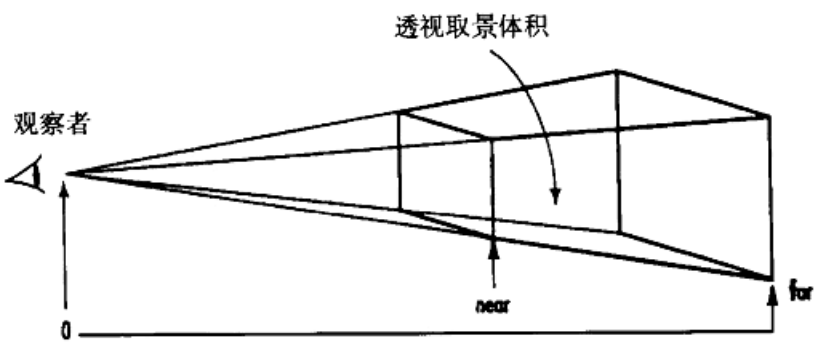


图 6-13 glFrustum 定义的透视投影

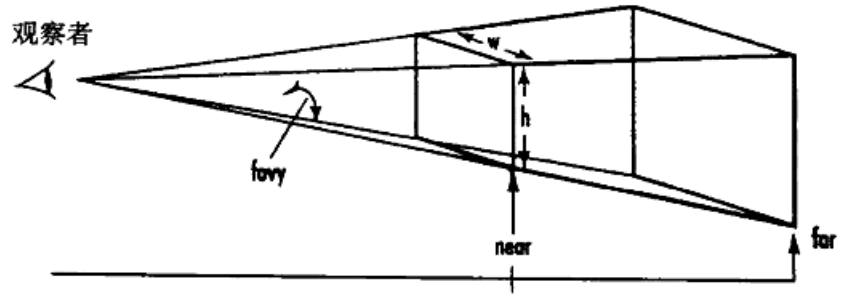


图 6-14　gluPerspective 定义的透视投影

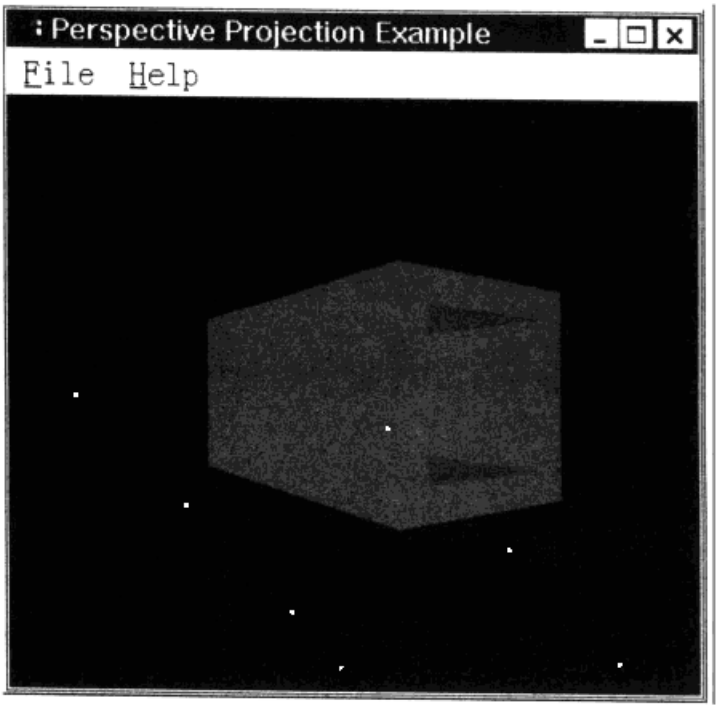


图 6-15　透视投影

程序 6-2 是使用透视投影的例子，如图 6-15 所示。

程序 6-2

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w/(GLfloat)h;
```

```
    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Produce the perspective projection
    gluPerspective(60.0f, fAspect, 1.0, 400.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }
```

6.5 一个例子

作为使用模型取景矩阵和透视投影的例子，模拟太阳、地球、月亮系统。在程序中使用了光照和明暗处理。

地球绕太阳转，月亮又绕着地球转。在观察者后面放置一个光源照亮太阳，然后移到太阳中心。从本程序可以看出，用 OpenGL 实现有真实效果的场景是很容易的。

程序 6-3 给出了设置投影和绘图的函数，如图 6-16 所示。

程序 6-3

```
// Change viewing volume and viewport.  Called when window is resized

void ChangeSize(GLsizei w, GLsizei h)
     {
     GLfloat fAspect;

     // Prevent a divide by zero
     if(h == 0)
          h = 1;

     // Set Viewport to window dimensions
     glViewport(0, 0, w, h);

     // Calculate aspect ratio of the window
     fAspect = (GLfloat)w/(GLfloat)h;

     // Set the perspective coordinate system
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
```

```
    // field of view of 45 degrees, near and far planes 1.0 and 425
    gluPerspective(45.0f, fAspect, 1.0, 425.0);

    // Modelview matrix reset
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }

// Called to draw scene
void RenderScene(void)
    {
    // Earth and Moon angle of revolution
    static float fMoonRot = 0.0f;
    static float fEarthRot = 0.0f;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Set light position before viewing transformation
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);

    // Set material color, Red
    // Sun
    glRGB(255, 255, 0);
    auxSolidSphere(15.0f);

    // Move the light after we draw the sun!
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Rotate coordinate system
    glRotatef(fEarthRot, 0.0f, 1.0f, 0.0f);

    // Draw the Earth
    glRGB(0,0,255);
```

```
glTranslatef(105.0f,0.0f,0.0f);
auxSolidSphere(15.0f);

// Rotate from Earth based coordinates and draw Moon
glRGB(200,200,200);
glRotatef(fMoonRot,0.0f, 1.0f, 0.0f);
glTranslatef(30.0f, 0.0f, 0.0f);
fMoonRot+= 15.0f;
if(fMoonRot > 360.0f)
     fMoonRot = 0.0f;

auxSolidSphere(6.0f);

// Restore the matrix state
glPopMatrix(); // Modelview matrix

// Step earth orbit 5 degrees
fEarthRot += 5.0f;
if(fEarthRot > 360.0f)
     fEarthRot = 0.0f;

// Flush drawing commands
glFlush();
}
```

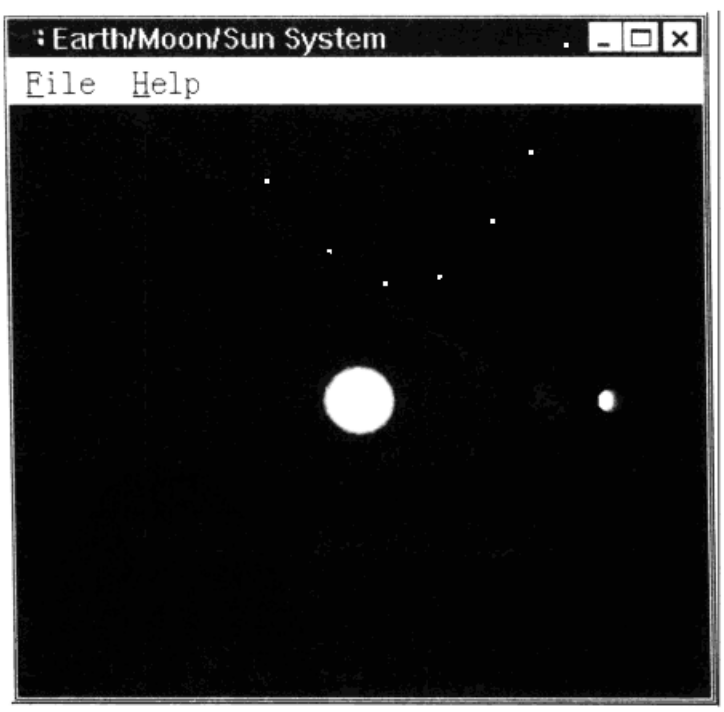


图 6-16　坐标变换的例子

6.6 相关函数

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);

void glLoadIdentity(void);

void glLoadMatrix{fd}(const TYPE *m);

void glMatrixMode(GLenum mode);

void glPushMatrix(void);

void glPopMatrix(void);

void glMultiMatrix{fd}(const TYPE *m);

void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);

void glTranslate{fd}(TYPE x, TYPE y, TYPE z);

void glScale{fd}(TYPE x, TYPE y, TYPE z);

void gluOrtho2D (GLdouble left, GLdouble right, GLdouble bottom, GLdouble top);

void gluPerspective (GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);

void gluLookAt (GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx, GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);

第七章　颜色和明暗处理

如果用计算机生成一种颜色，就要分别指定红、绿、蓝三种组分的强度。不同的颜色所对应的三种组分的强度组分值也不相同。在计算机屏幕后的电子枪把电子发射到屏幕背面，受电子轰击的屏幕有可以发射红、绿、蓝光的发光物质。所发射光的强度随电子束的强度变化，会集到屏幕上成为一个光点，计算机彩色显示器可以产生红、绿、蓝三种颜色的组合。

OpenGL支持如下两类颜色模式:

①RGBA颜色模式，用红、绿、蓝三个组分定义一个颜色。

②色彩指数模式，在绘图时，通过指定调色板中色彩的指数选择一个颜色。在调色板中，指定红、绿、蓝组分的强度定义一种颜色。

7.1　计算机显示的颜色

在本书第三章中，OpenGL调用glColor命令定义颜色的红、绿、蓝强度，即是在RGBA颜色模式下定义的颜色。

1. PC颜色硬件

从计算机图形硬件发展史上看，可以把硬件按性能分类为:

①Hercules图形卡，可以生成720×348像素的图像，每个像素只有开、关两种状态。只能绘制出单色图形。

②CGA (Color Graphics Adapter)，支持320×200像素，在屏幕上同时显示16种颜色中的四种。在使用两种颜色时分辨率可以达到640×200，用于简单PC游戏和商业软件。

③EGA (Enhanced Graphics Adapter)，支持25行彩色文本模式，支持640×350像素的16色图形，消除了CGA显示卡的闪烁问题，动画效果较好。可以运行真正的商业图形，甚至是三维图形。

④VGA(Vector Graphic Array)，支持640×480像素16色图形和320×200像素256色，速度远快于EGA显示卡。

256种颜色可以从有16M颜色的调色板中选取。

⑤SVGA的分辨率越来越高，由当初的800×600像素，发展到1024×768，1280×1024，所支持的颜色由256到3200～65000色，现在的24位显示卡可以显示16M颜色，分辨率1024×768。

Microsoft把OpenGL移植到Windows平台，正是为在PC上创建高端图形应用程序而设计的。尽管现在的Pentium和Pentium II PC的性能还不能与SGI专业图形工作站相比，但采用了3D图形加速功能的显示卡会大大提高PC的性能。可以预见，家用计算机做复杂模拟、设计3D游戏等功能的日子不会太远。

2. 计算机显示模式

Microsoft Windows在以下两个方面对PC图形界做了革命。

①创建了有大量用户的主流图形操作环境；

②使PC图形编程简便易行。

在Windows环境下，硬件由Windows显示设备驱动程序控制，而不必直接针对图形硬件写指令。用户可以调用一个API，由Windows与硬件打交道。Microsoft提供了流行显示卡的驱动程序包。硬件制造商提供最新的驱动程序，这些驱动程序可以在BBS或internet上找到。

3．屏幕分辨率

目前，PC机的屏幕分辨率从640×480到1280×1024像素，甚至于更高一些，但是屏幕分辨率通常并不是一个编写图形应用程序最重要的限制因素。即使是640×480的分辨率对大多数图形显示来说，也是足够的。比分辨率更为重要的是窗口的尺寸。在把所绘制图形的尺寸缩放到窗口尺寸时，可以考虑到可能出现的不同分辨率和窗口尺寸的组合。正确的应用程序在不同分辨率的屏幕上应该显示出大致相同的图像。加大屏幕显示的分辨率，用户看到的细节就越清楚。

4．颜色深度

加大屏幕的分辨率或者说使用的像素越来越多，可以得到图形更多的细节，也应该增加色彩以提高图像的清晰度。显示16M颜色的计算机屏幕要远比16色显示器好看得多。在编程时，常见的颜色深度有4位，8位和24位三种。所谓颜色深度是指计算机存储一个颜色所用的字位数目。

①4位颜色

这是彩色显示器开始出现时所采用的颜色模式。4位颜色模式支持16种颜色，代表从0到15种预定义颜色。仅用16色，很难改进图像的清晰度，所以多数质量要求较高的应用程序均不采用这一模式。

②8位颜色

这也是出现比较早的一种颜色模式。8位颜色模式支持256位颜色，代表从0到255位颜色调色板中的颜色，这些颜色可以从调色板的16M颜色中选取。在这一模式下，做抖动处理后的图像可以满足许多应用程序的需要。调色板中的每一种颜色可由指定红、绿、蓝三种组分的8个字位实现，每一组分强度从0到255。共有16M种选择。仔细选择这些颜色，在PC屏幕上可以得到接近照片质量的图形。

③24位颜色

这一颜色模式是目前一般计算机硬件条件下可以满足的，也是比较普遍采用的。24位颜色模式支持16M，红、绿、蓝三种组分分别对应8位颜色数据，使用这一模式需要在高分辨率时有大量内存支持，如1024×768分辨率要求2MB多内存。

④其他颜色深度

为节省内存、提高性能，许多显示卡还支持其他颜色模式。例如，为提高性能，一些显示卡支持32位颜色模式，这个模式有时称为真彩色模式。需要注意的是，32位颜色模式所支持的颜色并不比24位更多，但是可以通过把像素数据在32位地址边界对齐来提高绘图性能。这样，每个像素就浪费了8位。在32位Intel PC机上，内存地址可被32整除导致访问内存的速度加快。

另外两个颜色显示模式使用内存的效率也比较高。15位颜色模式，每5位分别存贮红、绿、蓝三种组分，每个像素可显示32768种不同颜色；16位模式在三种颜色组分之一加上一个字位(通常在绿色组分上加上一位)，支持65536种颜色。在绘图时，都可以获得24位图像

的效果。对大多数图像，很难分辨出16位与24位模式的差别。有时用16位模式得到的光滑明暗处理表面可以看到有一些条纹。

在编程时，在15位、16位模式设置颜色的方法与24位模式相同，即设置成三组8位数，在设定像素颜色之前，把这个24位彩色模式值缩放到与15位、16位模式相近的值。

7.2 选择颜色

颜色由三个颜色组分的正值组合值确定，可以把所有可能的颜色表述成一个RGB颜色空间。红、绿、蓝坐标类似于直角坐标x, y, z。对于24位彩色模式，每个轴的最大值均为255。原点在(0, 0, 0)，对应黑色；而(255, 255, 255)对应白色，每个轴的端点分别对应纯红、绿、蓝。图7-1为光滑明暗处理的彩色方块，方块的表面从一个角的黑色变到另一个角的白色。可以用方向键旋转这个方块，观察其颜色分布情况。

图7-1 一个彩色方块

1. 明暗处理

对于一点来说，定义了顶点的颜色，也就是该点的颜色。如果把一条线上的两个顶点赋予不同的颜色值，则线的颜色由明暗处理模式确定，明暗处理可以理解为从一种颜色到另一种颜色的过渡方法。OpenGL的明暗处理模式有两种，即光滑明暗处理和平直明暗处理模式。屏幕显示物体的颜色不仅受到设置颜色的OpenGL函数作用，也会因为在绘制场景时添加的特殊视觉效果，如光照、雾等而改变。

因为在RGB颜色空间上，两点对应两种颜色，而光滑明暗处理使颜色按两点间连线上点对应的颜色变化。图7-2为连接(0, 0, 0)，(255, 255, 255)连线上颜色的变化的趋势。

程序7-1 演示对三个顶点分别为红、绿、蓝的三角形所做的光滑明暗处理，如图7-3所示。

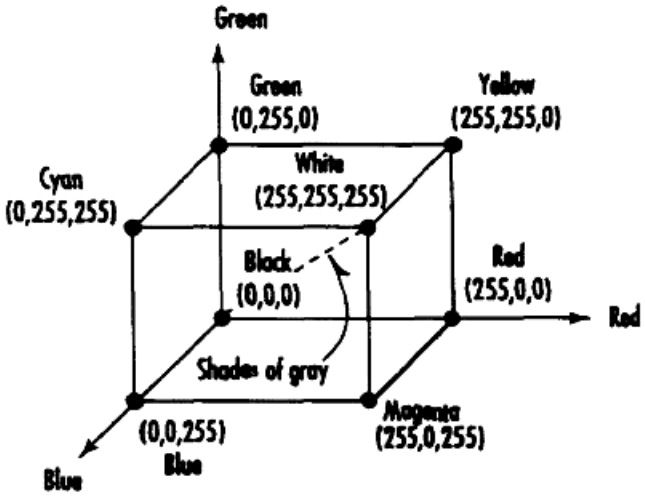


图7-2　RGB空间上颜色变化的趋势

程序7-1

```
// Called to draw scene
void RenderScene(void)
        {
        // Clear the window with current clearing color□汧氿慆
glClear(GL_COLOR_BUFFER_BIT);

        // Enable smooth shading
        glShadeModel(GL_SMOOTH);

        // Draw the triangle
        glBegin(GL_TRIANGLES);
                // Red Apex
                glColor3ub((GLubyte)255, (GLubyte)0, (GLubyte)0);
                glVertex3f(0.0f, 200.0f, 0.0f);

                // Green on the right bottom corner
                glColor3ub((GLubyte)0, (GLubyte)255, (GLubyte)0);
                glVertex3f(200.0f, -70.0f, 0.0f);

                // Blue on the left bottom corner
                glColor3ub((GLubyte)0, (GLubyte)0, (GLubyte)255);
                glVertex3f(-200.0f, -70.0f, 0.0f);
        glEnd();

        // Flush drawing commands
        glFlush();
        }
```

2. 设置明暗处理模式

程序7-1设置OpenGL的明暗处理模式。这也是OpenGL进行明暗处理的缺省模式，但是为使程序按设定的方式运行，应该调用这个函数。如果要采用平直明暗处理，调用glShadeModel函数时，参数应该为GL_FLAT，即

```
void glShadeModel (GL_FLAT);
```

在平直明暗处理模式下，不对几何要素内部做明暗处理。除了在glBegin/glEnd选用参数GL_POLYGON时，用第一个顶点的颜色填充内部外，均用最后一个顶点的颜色。

从上面的例子可以看出，无论多么复杂的多边形，OpenGL都会成功地对其内部做明暗处理。值得注意的是，尽管明暗处理的过程很复杂，但OpenGL会自动完成，而不必用户做任何干预。这样，大大减少了用户为复杂场景做明暗处理的工作量，也再一次体现了用OpenGL绘制三维图像的优越性。

虽然在不同的Windows的色彩设置下，程序7-1都能正确运行，但在24位彩色模式下显示的颜色过渡更光滑，如图7-3所示。

图7-3　三角形颜色过渡

7.3　Windows 环境下的调色板

1. 颜色匹配

在绘制某种RGB值颜色的像素时，Windows分别用一个8字位数定义红、绿、蓝三种组分。若计算机的屏幕显示属性为24位彩色模式，则按这三种组分定义的颜色显示该像素。在15位或16位模式下，Windows把24位颜色值传给显示驱动程序，使其在显示前，把这个颜色转换成15位或16位值。24位彩色模式的RGB空间每侧为255，而15位、16位模式为32或64。设备驱动程序把这个24位值与最相近的15,16位颜色相匹配，得到一个相近的颜色。

2. 抖动

在4位彩色模式下，仅能显示八种颜色，得到的图像质量很差。如果用Windows GDI的抖动操作处理所绘制的场景，可以使图像质量得到一定改善。抖动操作就是把不同的颜色放在一起复合成另一种颜色，Windows用抖动得到调色板中没有的颜色。

通常情况下，Windows不为OpenGL绘制的图像做抖动操作，可以调用下面的函数，允许使用该功能。

```
glEnable(GL_DITHER);
```

在8位或更高位数彩色模式下做抖动操作，图像质量得到的改善远不如16位模式下的效果明显，但是有时可以在8位或16位模式下大幅度提高图像的质量。

3．8位模式下调色板的优势

8位模式可以同时显示16M颜色中的256种，这256种颜色位于调色板中，所以精心挑选调色板中的颜色能显著提高图形质量。在Windows环境下，应用程序用CreatePalette函数创建一个由HPALETTE类型句柄标识的调色板。逻辑调色板包含256个条目，分别定义8位红、绿、蓝组分。在讲述如何创建调色板之前，先看一看8位彩色模式下，多任务应用程序是如何共享系统的唯一调色板的。

Windows的多任务功能允许屏幕上同时出现多个应用程序，因此，这些应用程序要共享系统调色板。假如一个应用程序改变了系统调色板，会扰乱其他窗口的颜色。为调整各应用程序对调色板的使用，Windows发出一系列信息。其他应用程序修改了系统调色板或窗口输入焦点而可能改变调色板时，会给应用程序发信息。

当应用程序接收键盘或鼠标输入焦点时，Windows给应用程序主窗口发出WM_QUERYNEWPALETTE信息，询问应用程序是否要实现一个新调色板，也就是说，把其调色板的条目拷贝到系统调色板上。为此，应用程序在要改动窗口的设备描述表中选择调色板，再调用RealisePalette函数。程序7-2是上述过程的具体实现。

程序7-2

```
// Windows is telling the application that it may modify
// the system palette.  This message in essance asks the
// application for a new palette.

case WM_QUERYNEWPALETTE:

        // If the palette was created.

        if(hPalette)
                {
                int nRet;

                // Selects the palette into the current device context
                SelectPalette(hDC, hPalette, FALSE);

                // Map entries from the currently selected palette to
                // the system palette.  The return value is the number
                // of palette entries modified.
                nRet = RealizePalette(hDC);
```

```
                    // Repaint, forces remap of palette in current window
                    InvalidateRect(hWnd,NULL,FALSE);

                    return nRet;
                    }
            break;

        // This window may set the palette, even though it is not the
        // currently active window.
        case WM_PALETTECHANGED:
                // Don't do anything if the palette does not exist, or if
                // this is the window that changed the palette.
                if((hPalette != NULL) && ((HWND)wParam != hWnd))
                        {
                        // Select the palette into the device context
                        SelectPalette(hDC,hPalette,FALSE);

                        // Map entries to system palette
                        RealizePalette(hDC);

                        // Remap the current colors to the newly realized palette
                        UpdateColors(hDC);
                        return 0;
                        }
                break;
```

Windows发出实现调色板的另一个信息是WM_PALETTECHANGED。这一信息发给可以实现调色板但没有当前焦点的窗口。这一信息发出后，用户要检查wParam的值。若wParam有接受信息的当前窗口句柄，则已经处理过WM_QUREYNEWPALETTE信息，调色板不必再实现。

在处理这两个信息前，hPalette的值不能为NULL。若应用程序不在8位彩色模式运行，不必用这些函数创建或实现调色板。

7.4 创建调色板

1. 是否需要调色板?

在选择了一个像素格式后，可以调用DescribePixelFormat( )函数，检验由DescribePixelFormat( )函数返回的PIXELFORMATDESCRIPTOR结构中dwFlags的值。若给PFD_NEED_PALETTE位赋值，则在应用程序中要创建调色板。

程序7-3给出做上述检验的程序段:

程序7-3

```
PIXELFORMATDESCRIPTOR pfd;        // Pixel Format Descriptor
```

```
int nPixelFormat;                    // Pixel format index

// Get the pixel format index and retrieve the pixel format description
nPixelFormat = GetPixelFormat(hDC);
DescribePixelFormat(hDC, nPixelFormat, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

// Does this pixel format require a palette?  If not, do not create a
// palette and just return NULL
if(!(pfd.dwFlags & PFD_NEED_PALETTE))
        return NULL;
// palette creation code here
```

2. 调色板的结构

在创建调色板前，要给LOGPALETTE结构分配内存，这一结构存贮着描述调色板的信息，再把它传递给Win32函数CreatePalette，LOGPALETTE的定义如下：

```
typedef struct tagLOGPALETTE { // lgpl
    WORD                palVersion;
    WORD                palNumEntries;
    PALETTEENTRY        palPalEntry[1];
    } LOGPALETTET
```

前两个成分分别为调色板名称的版本(通常为0x300)以及所包含的颜色条目(对于8位模式为256)，每个条目定义为一个包含该条目RGB组分的PALETTEENTRY结构。下面的程序段为LOGPALETTE分配内存：

```
*pPal;                  // Pointer to memory for logical palette
// Allocate space for a logical palette structure plus all the palette entries
pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +nColors*sizeof(PALETTEENTRY));
```

其中nColors为调色板中包含的颜色数目，PALETTEENTRY结构的定义如下：

```
typedef struct tagPALETTEENTRY  { // pe
        BYTE                    peRed;
        BYTE                    peGreen;
        BYTE                    peBlue;
        BYTE                    peFlags
        } PALETTEENTRY
```

其中peRed，peGreen，peBlue分别为代表每一个颜色组分的8位值。这样，共有256个条目的调色板中每一条目都可以存贮一个24位颜色信息，peFlags成分描述使用条目的高级用法，在OpenGL中，可以设为NULL。

在Windows环境下，OpenGL的RGBA颜色模式仅支持3-3-2调色板。由DescribePixelFormat函数返回PIXELFORMATDESCRIPTOR结构中，cRedBits，cGreenBits和cBluebits成分分别定义表示每个组分数目为3,3,2。cRedshift，cGreenshift和cBlueShift定义每个组分左移的数目。

所谓3-3-2调色板是指在8位彩色模式下，用于存贮红、绿、蓝组分的位数分别为3,3,2。

在RGB颜色空间上，红、绿、蓝的最大值分别为7，7，3。调色板中的颜色由24位模式8-8-8颜色调整到3-3-2颜色。

程序7-4所示为创建一个调色板的过程。

程序7-4

```
// If necessary, creates a 3-3-2 palette for the device context listed.
HPALETTE GetOpenGLPalette(HDC hDC)
        {
        HPALETTE hRetPal = NULL; // Handle to palette to be created
        PIXELFORMATDESCRIPTOR pfd;       // Pixel Format Descriptor
        LOGPALETTE *pPal;                        // Pointer to memory for logical palette
        int nPixelFormat;                        // Pixel format index
        int nColors;                             // Number of entries in palette
        int i;                                   // Counting variable
        BYTE RedRange,GreenRange,BlueRange;// Range for each color entry (7,7,and 3)

        // Get the pixel format index and retrieve the pixel format description
        nPixelFormat = GetPixelFormat(hDC);
        DescribePixelFormat(hDC, nPixelFormat, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

        // Does this pixel format require a palette?  If not, do not create a
        // palette and just return NULL
        if(!(pfd.dwFlags & PFD_NEED_PALETTE))
                return NULL;

        // Number of entries in palette.  8 bits yeilds 256 entries
        nColors = 1 << pfd.cColorBits;

        // Allocate space for a logical palette structure plus all the palette entries
        pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE)+nColors*sizeof(PALETTEENTRY));

        // Fill in palette header
        pPal->palVersion = 0x300;                // Windows 3.0
        pPal->palNumEntries = nColors; // table size

        // Build mask of all 1's.  This creates a number represented by having
        // the low order x bits set, where x = pfd.cRedBits, pfd.cGreenBits, and
        // pfd.cBlueBits.
        RedRange = (1 << pfd.cRedBits) -1;
        GreenRange = (1 << pfd.cGreenBits) - 1;
        BlueRange = (1 << pfd.cBlueBits) -1;

        // Loop through all the palette entries
```

```
for(i = 0; i < nColors; i++)
        {
        // Fill in the 8-bit equivalents for each component
        pPal->palPalEntry[i].peRed = (i >> pfd.cRedShift) & RedRange;
        pPal->palPalEntry[i].peRed = (unsigned char)(
                (double) pPal->palPalEntry[i].peRed * 255.0 / RedRange);

        pPal->palPalEntry[i].peGreen = (i >> pfd.cGreenShift) & GreenRange;
        pPal->palPalEntry[i].peGreen = (unsigned char)(
                (double)pPal->palPalEntry[i].peGreen * 255.0 / GreenRange);

        pPal->palPalEntry[i].peBlue = (i >> pfd.cBlueShift) & BlueRange;
        pPal->palPalEntry[i].peBlue = (unsigned char)(
                (double)pPal->palPalEntry[i].peBlue * 255.0 / BlueRange);

        pPal->palPalEntry[i].peFlags = (unsigned char) NULL;
        }

// Create the palette
hRetPal = CreatePalette(pPal);

// Go ahead and select and realize the palette for this device context
SelectPalette(hDC,hRetPal,FALSE);
RealizePalette(hDC);

// Free the memory used for the logical palette structure
free(pPal);

// Return the handle to the new palette
return hRetPal;
}
```

3. 对调色板的操作

在创建绘图描述表并把它设为当前值之前，必须创建并实现调色板。在设定像素格式后，需要一个设备描述表，然后返回调色板的句柄(如果有必要，使用调色板)。下面的程序给出创建、关闭窗口的过程。与对绘图描述表的操作相似，只是要考虑到可能存在调色板。

```
        // Window creation, setup for OpenGL
        case WM_CREATE:
                // Store the device context
                hDC = GetDC(hWnd);

                // Select the pixel format
```

```
            SetDCPixelFormat(hDC);

            // Create the palette if needed
            hPalette = GetOpenGLPalette(hDC);

            // Create the rendering context and make it current
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);
            break;

    // Window is being destroyed, cleanup
    case WM_DESTROY:
            // Deselect the current rendering context and delete it
            wglMakeCurrent(hDC,NULL);
            wglDeleteContext(hRC);

            // If a palette was created, destroy it here
            if(hPalette != NULL)
                    DeleteObject(hPalette);

            // Tell the application to terminate after the window
            // is gone.
            PostQuitMessage(0);
            break;

    // Window is resized.
    case WM_SIZE:
            // Call our function which modifies the clipping
            // volume and viewport
            ChangeSize(LOWORD(lParam), HIWORD(lParam));
            break;
```

值得注意的是，并非256个调色板条目都映射到系统调色板中，Windows保留20个用于包含16个标准VGA/EGA颜色的静态系统颜色，使标准窗口成分(标题、按钮等)在应用程序改变系统调色板时不发生变化。在应用程序实现其调色板时，这几个颜色不会被覆盖。

7.5 色彩指数模式

除了前面一直使用的RGBA颜色模式，OpenGL也支持色彩指数模式，在这一模式下，通过指定颜色数组的指标来定义颜色，而不是定义RGB三组分。色彩指数模式与RGBA模式不能同时使用，也就是说，在色彩指数模式下，不能得到真彩色设备提供的所有颜色。在某些OpenGL

实现中，色彩指数调色板可以达到4096个条目，而Microsoft实现仅支持256色。

1．色彩指数模式优点

①可以更多地控制调色板；

②在应用程序中，采用颜色标识空间变量分布；

③比8位模式RGBA快。

2．色彩指数模式缺点

①不支持OpenGL的某些特殊效果，如光照、明暗处理、雾、反走样、α混合操作等；

②选择颜色受到限制。

在8位彩色模式下，3-3-2调色板限制了所选择颜色，而色彩指数模式不受限制。图7-4所示的三角形由黑色光滑过渡到亮红色，8位彩色模式下的3-3-2调色板是不可能做到的。

图7-4　使用色彩指数模式

3．使用色彩指数模式

为使用色彩指数模式，要把PIXELFORMATDESCRIPTOR结构的iPixelType成分设置成PFD_TYPE_COLORINDEX。首先创建一个调色板，程序7-5为创建调色板的程序段。

程序7-5

```
// Creates a color ramp from black to bright red

HPALETTE GetRedPalette(HDC hDC)
        {
        HPALETTE hRetPal = NULL; // Handle to palette to be created
        LOGPALETTE *pPal;                       // Pointer to memory for logical palette
        int i;                                          // Counting variable

        // Allocate space for a logical palette structure plus all the palette entries
        pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +256*sizeof(PALETTEENTRY));

        // Fill in palette header
        pPal->palVersion = 0x300;               // Windows 3.0
```

```
    pPal->palNumEntries = 256;                  // table size

    // Loop through all the palette entries, creating a graduated red
    // palette containing only shades of red
    for(i = 0; i < 256; i++)
            {
            pPal->palPalEntry[i].peRed = i;          // Red intensity from 0 to 255
            pPal->palPalEntry[i].peGreen = 0;
            pPal->palPalEntry[i].peBlue = 0;
            pPal->palPalEntry[i].peFlags = (unsigned char) NULL;
            }

    // Create the palette
    hRetPal = CreatePalette(pPal);

    // Go ahead and select and realize the palette for this device context
    SelectPalette(hDC,hRetPal,FALSE);
    RealizePalette(hDC);

    // Free the memory used for the logical palette structure
    free(pPal);

    // Return the handle to the new palette
    return hRetPal;
    }
```

程序7-6绘制三角形，设置顶点颜色并选用光滑明暗处理。

程序7-6

```
// Called to draw scene
void RenderScene(void)
    {
    // Enable smooth shading
    glShadeModel(GL_SMOOTH);

    // Set clear color to first entry in palette (black)
    glClearIndex(0.0f);

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Draw the triangle
```

```
glBegin(GL_TRIANGLES);
        // Darkest Red Apex (black)
        glIndexi(0);
        glVertex3f(0.0f, 200.0f, 0.0f);

        // Brightest red bottom corners
        glIndexi(255);
        glVertex3f(200.0f, -70.0f, 0.0f);
        glVertex3f(-200.0f, -70.0f, 0.0f);
glEnd();

// Flush drawing commands
glFlush();
}
```

7.6 相关函数

```
void glClearIndex(GLfloat c);
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glColorMask(GLboolean red, GLboolean green, GLboolean blue, GLboolean alpha);
void glIndexMask(GLuint mask);
void glLogicOP (GLenum opcode);
void glShadeModel(GLenum mode);
void glIndex{dfis} (TYPE c);
```

第八章　光照处理

尽管前面几章介绍了在三维空间绘制复杂物体的初步知识，但是应该看到，在真实存在的物体上看到的颜色要远比前几章中所提到的颜色丰富得多。为绘制更逼真的场景，就需要对场景做特殊的处理，使场景增加一些特殊的视觉效果。本章及后面的几章将主要讲述如何实现这些特殊效果。在这些特殊效果中，光照处理是一个非常重要的内容。本章的内容就是讲述如何利用光照处理得到更逼真的图像。在实际生活中观察物体，物体的颜色会随光照的明暗以及光源的颜色而发生变化，可能会闪闪发光；也可能暗淡无光；从某一角度看一个被照亮的物体会出现阴影。下面讲述实现这些效果的具体步骤。

8.1　光源种类

OpenGL 用光照处理可以得到非常逼真的图像。除了物体可能自身发光外，还有另外三种光源：

①泛光，不发自某一特定方向，光线在壁面或场景中经过多次反射而变成没有方向的光线。被泛光照亮的物体在所有方向、所有表面都被均匀照射。可以把本书前面所举例子都看成被一个亮的泛光照射而成的，因为无论在怎样的方向、视角，物体都是可见的且均匀着色。

②漫反射光：漫反射光来自某一特定方向，但经物面均匀反射。尽管光线经过均匀反射，被直接照射的物面也要比倾斜一个角度照射的情况要亮一些。 典型漫反射的例子有荧光灯光源，照射在窗户的阳光等。漫反射光示意图如图 8-1 所示。

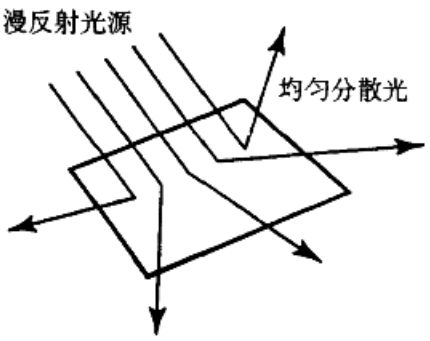


图 8-1　漫反射光示意图

③镜面光：镜面光也是有方向的，在某一特定方向最亮。高亮度的镜面光会在物面上形成一个亮斑，称为镜面高光，聚光灯和太阳光是镜面光，如图 8-2 所示。

当然任何一个光源都不可能只由上面提到的单一类型光源构成，而是由不同强度的三种光源组成的，例如，红色激光束基本是红色镜面光源，但是在灰尘或烟雾中会散射，因此，在整个房间都可以看到，也就是说光源中存在着漫反射光成分；另外，光束照射到物体表面

后，在室内都可以看到，说明存在泛光成分。

上述三种光源由 RGBA 值定义颜色，描述红、绿、蓝光的相对强度。对于红色激光束，RGBA 组分见表 8-1。

值得注意的是，在红色激光束没有绿、蓝光成分。从红色组分可以看出，这三种光源的强弱顺序是，镜面光最强，漫反射光较弱，泛光最弱。

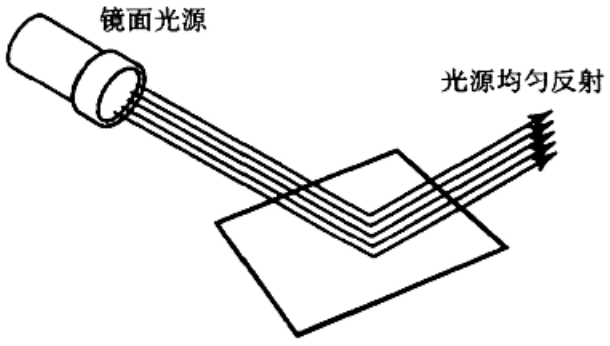


图 8-2　镜面光示意图

表 8-1　红色激光束的颜色和光强

	红	绿	蓝	α
镜面	0.99	0.0	0.0	1.0
漫反射	0.10	0.0	0.0	1.0
泛光	0.05	0.0	0.0	1.0

8.2　材质属性

真实存在的物体本身都有颜色。物体的颜色是由其反射光的波长决定的。实际的光线绝大多数由白光和其他颜色混合而成。物体在白光下更能显出其固有颜色。对应计算机图形学，进行光照处理时，不仅要设置适当的光源属性，也要选择合适的材质属性，否则，无法得到正确的效果。

在光照条件下，不能说多边形有某种颜色，而要把多边形描述成由有一定反射特性的材质构成的。例如，不把多边形定义为红色，而是由反射绝大多数红光物质构成的。谈到物面，就要定义对泛光、漫反射、镜面光源的反射特性。另外需要定义的属性是在物体表面发射光的属性，如夜光表等。几何要素每一顶点的 RGBA 值是由泛光、漫反射、镜面光分别与材质对泛光、漫反射、镜面反射属性相乘得到的净效果决定的。如何设定光照和材质属性以得到某一视觉效果，需要经过一定时间的练习才能掌握。

修改程序 8-1 中有关光照、材质的参数（见图 8-3），可以使读者体验到经过光照处理后场景的真实性，以及如何在程序中为得到某一视觉效果而选择适当的参数。程序 8-1 是一个 C++程序，请参见与本书配套的软盘。

1．计算泛光效果

如果光源的泛光 RGBA 值为(0.5，0.5，0.5)，而材质对泛光的反射属性 RGB 值为(0.5,1.0,0.5)，得到的净效应为：

(0.50*0.50，0.5*1.0，0.50*0.50)=(0.25，0.5，0.25)

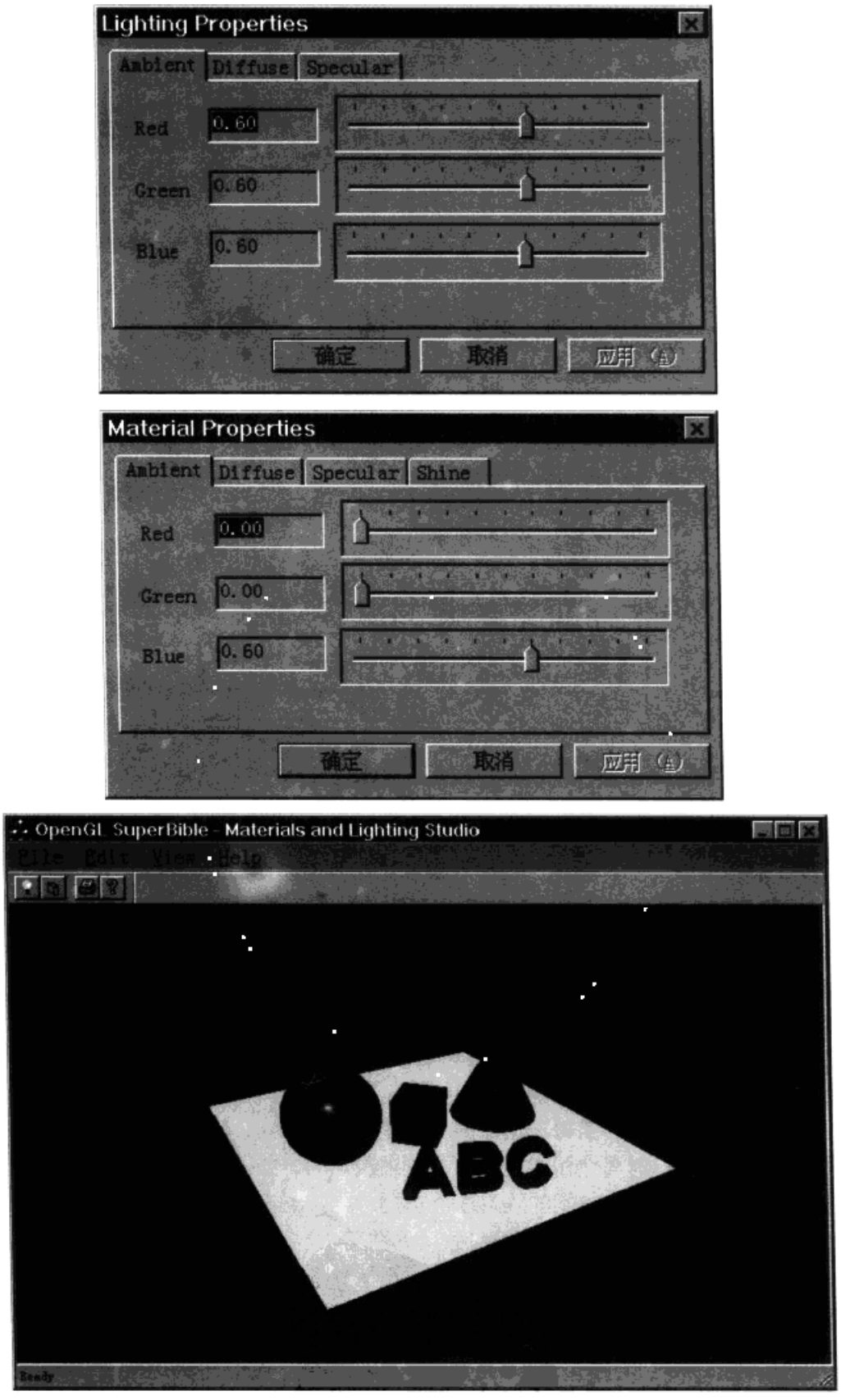

Lighting Properties
Ambient
Diffuse
Specular
Red
0.60
Green
0.60
Blue
0.60
确定
取消
应用 (A)
Material Properties
Ambient
Diffuse
Specular
Shine
Red
0.00
Green
0.00
Blue
0.60
确定
取消
应用 (A)
OpenGL SuperBible - Materials and Lighting Studio
File Edit View Help
ABC
Ready

图 8-3　光照处理的场景

由此可见，材质颜色组分决定了对入射光的反射率。本例中，材质属性的红色组分为0.5，即反射一半入射光的红色组分。

2．漫反射、镜面反射的计算

漫反射、镜面反射的 RGB 组分与材质属性作用的方式与泛光相同。然而，漫反射光、镜面反射光都是有方向的，物面的光强随物面与光源间的角度发生变化。

完成上述计算后，要把这三种效应相叠加，若某一颜色组分值大于 1.0，则应该限制在1.0。

通常，光源及材质的泛光和漫反射组分相同，对物体颜色的影响最大。镜面光的材质属性趋于亮灰或白色。镜面反射组分与入射光角度有关，物体上的镜面高光通常为白色。

8.3 在场景中加入光照

从计算机图形学的角度看，对三维场景做光照处理是非常复杂的，但是采用 OpenGL 应用程序解决这一问题时，仅需要掌握光照的基本概念以及进行光照处理的基本步骤，而不必与复杂的数学公式、算法打交道。尽管如此，对初学者来说，这部分内容仍是比较复杂的，所以，下面将结合几个 OpenGL 光照处理的程序说明在 OpenGL 中使用光照的技巧。

1．允许做光照处理

在缺省情况下，OpenGL 是不对场景进行光照处理的。如果要做光照处理，就要调用

```
glEnable(GL_LIGHTING);
```

允许在 OpenGL 中做光照计算，但要在光照计算之前，为场景中每一顶点指定光源和材质属性，否则，场景上不会有任何显示。

2．设置光照模型

在允许光照计算后，需要设置光照模型。调用 glLightModel 函数可以设定影响光照模型的三个参数。GL_LIGHT_MODEL_AMBIENT 是这三个参数中的一个，定义均匀照亮所有物体上各侧面的全景泛光，下面的程序段指定强白光。

```
// Bright white light
GLfloat ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };

// Lighting stuff
glEnable(GL_LIGHTING);              // Enable lighting

// Set light model to use ambient light specified by ambientLight
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,ambientLight);
```

glLightModelfv 把 RGBA 值赋给所要修改或设定的第一个参数，缺省的全景泛光 RGBA 值为(0.2，0.2，0.2，0.1)。其余的光照模型参数要设定的参数用于确定前、后面是否要做光照计算，计算镜面反射的角度等。

3．设置材质属性

即使有了一个泛光光源，也要设置一些材质的反射光属性。在指定多边形前，首先要调用 glMaterial 函数。如下面程序段所示：

```
GLfloat gray[] = {0.75f, 0.75f, 0.75f, 1.0f};
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);

glBegin(GL_TRIANGLES);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(0.0f, 15.0f, 30.0f);
    glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

glMaterialfv 的第一个参数指定要使用材质属性的面(GL_FRONT, GL_BACK 或者 GL_FRONT_AND_BACK 之一)；第二个参数为要改变的材质属性，本例中把泛光和漫反射属性设成相同的值；最后一个参数为设置上述属性的包含 RGBA 值的数组。调用 glMaterial 函数后，场景中的所有几何要素均受到影响。

在多数情况下，泛光和漫反射的值相同。在不使用镜面光源时，不要设置镜面反射属性，否则，对每一多边形都要定义 RGBA 值并调用 glMaterial 函数。

设置材质属性的另一方法称为颜色跟踪法，即仅用 glColor 函数就可以为 OpenGL 设置材质属性。下面的调用允许使用颜色跟踪方法。

```
glEnable(GL_COLOR_MATERIAL);
```

然后用 glColorMaterial 指定随着 glColor 设定颜色而变化的材质参数。例如，用颜色跟踪法设置多边形前面的泛光和漫反射光属性，可以调用

```
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);
```

把前面的程序段改为

```
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);

// Front material ambient and diffuse colors track glColor
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, gray);

    glColor3f(0.75f, 0.75f, 0.75f);
glBegin(GL_TRIANGLES);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(0.0f, 15.0f, 30.0f);
    glVertex3f(0.0f, 0.0f, -56.0f);
glEnd();
```

虽然看上去程序变长了，但实际上节省了很多行程序，并且随着多边形数量加大，会节省大量时间。

程序 8-2 为加入到 SetupRc 函数的程序段，设置强泛光及材质属性。如图 8-4 所示。

程序 8-2

```
// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for the scene.
void SetupRC()
```

```
{
// Light values
GLfloat  ambientLight[] = { 1.0f, 1.0f, 1.0f, 1.0f };
// Enable lighting
glEnable(GL_LIGHTING);

glEnable(GL_DEPTH_TEST);// Hidden surface removal
glEnable(GL_CULL_FACE);       // Do not calculate inside of jet
glFrontFace(GL_CCW);          // Counter clock-wise polygons face out

// Enable color tracking
glEnable(GL_COLOR_MATERIAL);

// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Light blue background
glClearColor(0.0f, 0.0f, 0.5f, 1.0f );
}
```

如果把泛光强度减小一半时，得到的效果是完全不同的，如图 8-5 所示。此时可以看出，减少场景泛光强度，图像会变暗。这可用来模拟遮挡直射光源在另一个物体上生成阴影时的效果。

图 8-4　使用泛光光源

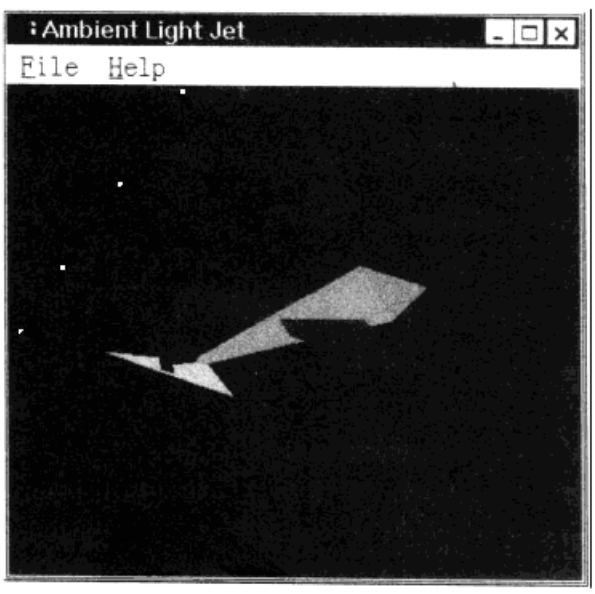


图 8-5　降低泛光光源强度

4. 使用光源

在绝大多数 OpenGL 应用程序中，为得到更逼真的图像，仅用泛光光源是不够的，还要同时使用其他种类的光源。这些光源的属性不仅有 RGBA 值强度，还有方向、位置等参数。它们对场景的视觉效果影响很大。

OpenGL 最多支持 8 个在场景中或置于取景体积外的相互独立的光源。可以把光源放在无穷远处，得到平行光线，也可以放在场景内，还可以使用聚光灯。

在 OpenGL 中使用光源，就要指出其方向和位置。光源可以照亮所有方向，也可以是某一方向。来自光源的光线会与构成物面的多边形成一个角度。OpenGL 在计算明暗效果时要用到这些角度。这些角度也与光照和材质属性一起用来计算多边形的颜色。

得到多边形各顶点颜色，再做光滑明暗处理，就可以获得光照效果。

从编程角度来看，多边形是由一系列顶点构成的，只有确定了曲面的法线方向才能确定该点与入射光线的夹角。法线是指定顶点(或多边形)方向的矢量，下面的程序段定义一个三角形的法线:

```
glBegin(GL_TRIANGLES);
    glNormal3f(0.0f, -1.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 60.0f);
    glVertex3f(-15.0f, 0.0f, 30.0f);
    glVertex3f(15.0f, 0.0f, 30.0f);
glEnd();
```

因为三角形位于 y=0 的平面上，法向矢量很容易给出。glNormal3f 函数定义三角形法向矢量的三个坐标分量，与三角形三个顶点的法方向相同。OpenGL 在光照处理中使用的法向矢量必须是单位矢量，即矢量的长度为 1 个长度单位。把法向矢量变成单位长度法线的过程称为归一化，即把矢量的三分量分别除以矢量的长度。这一过程可由 OpenGL 自动完成，即调用

```
glEnable(GL_NORMALIZE);
```

如果调用上述函数，会降低 OpenGL 的绘图性能，因此，最好用户在给定法线时，就是归一化处理后的值，而不用 OpenGL 完成这一工作。下面的程序段可以完成归一化处理。

```
// Reduces a normal vector specified as a set of three coordinates,
// to a unit normal vector of length one.
void ReduceToUnit(float vector[3])
    {
    float length;

    // Calculate the length of the vector
    length = (float)sqrt((vector[0]*vector[0]) +
                         (vector[1]*vector[1]) +
                         (vector[2]*vector[2]));

    // Keep the program from blowing up by providing an exceptable
    // value for vectors that may calculated too close to zero.
    if(length == 0.0f)
        length = 1.0f;

    // Dividing each element by the length will result in a
```

```
// unit normal vector.
vector[0] /= length;
vector[1] /= length;
vector[2] /= length;
}
```

5. 找到法线

如果多边形不在与坐标轴平行的平面上，可以用该多边形在一个平面内的三点定义两个矢量 **V1**，**V2**，求叉积 **V1×V2**，所得的矢量即为该多边形的法向矢量。做叉乘运算的程序如下：

```
// Points p1, p2, & p3 specified in counter clock-wise order
void calcNormal(float v[3][3], float out[3])
    {
    float v1[3],v2[3];
    static const int x = 0;
    static const int y = 1;
    static const int z = 2;

    // Calculate two vectors from the three points
    v1[x] = v[0][x] - v[1][x];
    v1[y] = v[0][y] - v[1][y];
    v1[z] = v[0][z] - v[1][z];

    v2[x] = v[1][x] - v[2][x];
    v2[y] = v[1][y] - v[2][y];
    v2[z] = v[1][z] - v[2][z];

    // Take the cross product of the two vectors to get
    // the normal vector which will be stored in out
    out[x] = v1[y]*v2[z] - v1[z]*v2[y];
    out[y] = v1[z]*v2[x] - v1[x]*v2[z];
    out[z] = v1[x]*v2[y] - v1[y]*v2[x];

    // Normalize the vector (shorten length to one)
    ReduceToUnit(out);
    }
```

6. 设置光源

在完成上面的各步骤后，下面该是打开光源的时候了。程序 8-3 所示的 SetupRC( )函数创建一个 GL_LIGHT0 光源并把它放在观察点稍后的左上角处，分别用 ambientLight[]，diffuseLight[]数组定义其泛光、漫反射光源强度，由此，得到一个中等强度白光光源：

```
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
```

```
.....
// Setup and enable light 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
```

光源的位置由下面的程序给定:

```
GLfloat    lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f };
.....
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

lightPos[]数组给出光源的位置。最后一个值是缩放系数，光源的实际位置为前三个坐标分量除以这个系数得到的三维坐标值。该数组的最后一个值为 1.0，表明该数组前三个坐标分量就是光源的位置。若该值为 0.0，则光源位于沿 x,y,z 三个分量所定义矢量的无穷远处。最后，调用 glEnable 函数，允许使用 LIGHT0 光源:

```
glEnable(GL_LIGHT0);
```

程序 8-3

```
// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
    {
    // Light values and coordinates
    GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    GLfloat    lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f };

    glEnable(GL_DEPTH_TEST); // Hidden surface removal
    glFrontFace(GL_CCW);          // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);       // Do not calculate inside of jet

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Setup and enable light 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glEnable(GL_LIGHT0);
    // Enable color tracking
    glEnable(GL_COLOR_MATERIAL);

    // Set Material properties to follow glColor values
```

```
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Light blue background
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}
```

7. 设置材质属性

下面的程序段采用颜色跟踪法定义多边形前表面的泛光和漫反射属性。

```
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);

// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// Light blue background
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}
```

图 8-6 所示为程序的输出结果。用方向键进行旋转，可以看到在光照条件下的物体表面有很强的明暗效果。

图 8-6 在场景中使用一个光源

8. 光照效果

尽管在上面的例子中仅用到了泛光和漫反射光源，也可以感受到光照带来的视觉效果。可以看到物面的明暗、判断出光线的方向。这种光源可以用于材质、布料等色彩比较平淡的物体建模，但是在模拟金属表面时，就要用到镜面光源和材质的镜面反射属性，这样，才能逼真地进行模拟。

①镜面高光

镜面光照和材质属性可以给物体表面增加光泽，使物体的颜色偏白。相对于观察点，很大入射光角度会产生镜面高光。此时，几乎所有打在物面上的光线都被反射。在阳光下红色光泽球上会出现白色亮光就是一个很好的例子。

②镜面光

在光源中可以很方便地加入镜面分量，其程序实现如下所示：

```
// Light values and coordinates
GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
GLfloat    lightPos[] = { -50.f, 50.0f, 100.0f, 1.0f };

glEnable(GL_DEPTH_TEST); // Hidden surface removal
glFrontFace(GL_CCW);          // Counter clock-wise polygons face out
glEnable(GL_CULL_FACE);       // Do not calculate inside of jet

// Enable lighting
glEnable(GL_LIGHTING);

// Setup and enable light 0
glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
glEnable(GL_LIGHT0);
```

specular[]数组定义一个很亮的白色镜面分量，用于模拟阳光。

```
glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
```

把镜面分量加入到GL_LIGHT0光源中，但是经过上述改动后，程序显示的结果并没有任何差别，这是因为还没有定义材质的镜面反射属性。

③镜面反射

为材质属性中加入镜面反射与在光源中加入镜面分量一样容易。下面的程序段用于在材质属性中加入镜面反射。

```
// Light values and coordinates
GLfloat  specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };
// Enable color tracking
glEnable(GL_COLOR_MATERIAL);

// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// All materials hereafter have full specular reflectivity
// with a high shine
```

```
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
glMateriali(GL_FRONT,GL_SHININESS,128);
```

specref[]数组的分量均为 1.0，即几乎全部反射入射的镜面光，其后所有的多边形均有这一属性。

④镜面指数

在上面的例子中很强的镜面光和反射率使照射的物面成为灰、白色。为减弱这一效果，在定义镜面分量后，调用

```
glMateriali(GL_FRONT,GL_SHININESS,128);
```

其中 GL_SHININESS 属性设置材质的镜面分量、镜面高光的大小和集中程度。0 值对应镜面高光不会聚，在整个多边形上产生亮光。加大这个值会缩小高光的尺寸、加大高光会集程度，以至最终得到一个亮斑。该值越大，表面就越亮。对所有 OpenGL 实现，GL_SHININESS 的取值范围在 0～128 之间。下面的程序段(程序 8-4)中加入了镜面反射属性分量，输出结果如图 8-7 所示。

程序 8-4

```
// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
    {
    // Light values and coordinates
    GLfloat  ambientLight[] = { 0.3f, 0.3f, 0.3f, 1.0f };
    GLfloat  diffuseLight[] = { 0.7f, 0.7f, 0.7f, 1.0f };
    GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat    lightPos[] = { 0.0f, 150.0f, 150.0f, 1.0f };
    GLfloat  specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };

    glEnable(GL_DEPTH_TEST);// Hidden surface removal
    glFrontFace(GL_CCW);          // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);       // Do not calculate inside of jet

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Setup and enable light 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glEnable(GL_LIGHT0);

    // Enable color tracking
```

```
glEnable(GL_COLOR_MATERIAL);

// Set Material properties to follow glColor values
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

// All materials hereafter have full specular reflectivity
// with a high shine
glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
glMateriali(GL_FRONT,GL_SHININESS,128);

// Light blue background
glClearColor(0.0f, 0.0f, 1.0f, 1.0f );
}
```

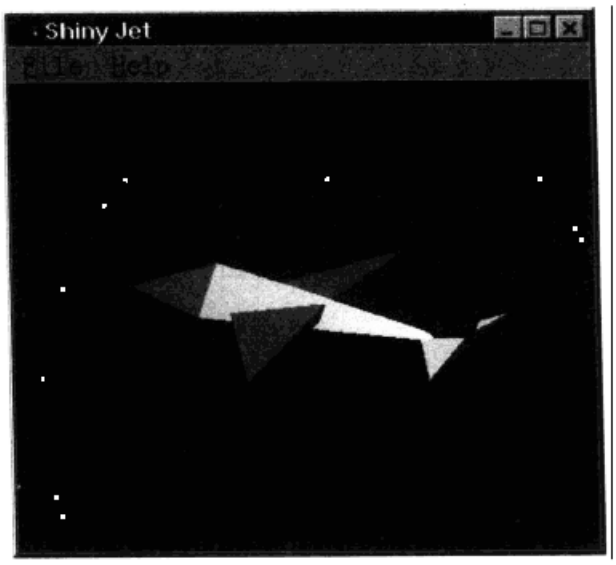


图 8-7　镜面高光

⑤法线平均

对于如图 8-8 所示的表面，相邻的多边形上法向矢量变化较大，这时，处理两个多边形交界处的法线应该取这两个多边形法线的平均值，这一过程称为法线平均。这样，OpenGL 的明暗处理在两多边形之间的变化才不会太剧烈。

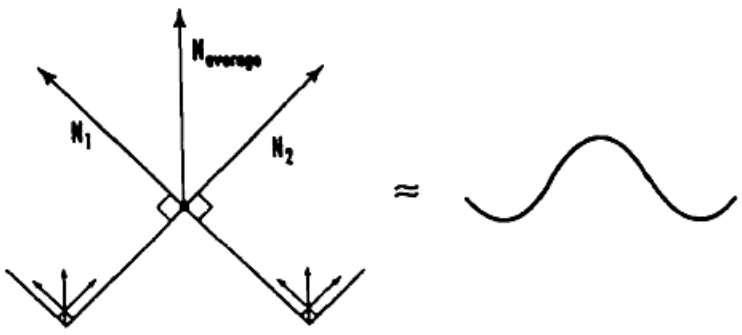


图 8-8　表面法线平均示意图

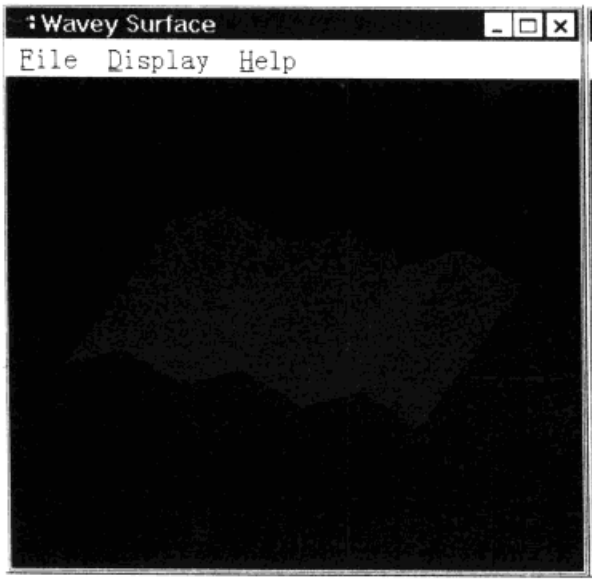


图 8-9 没有法线平均的图形

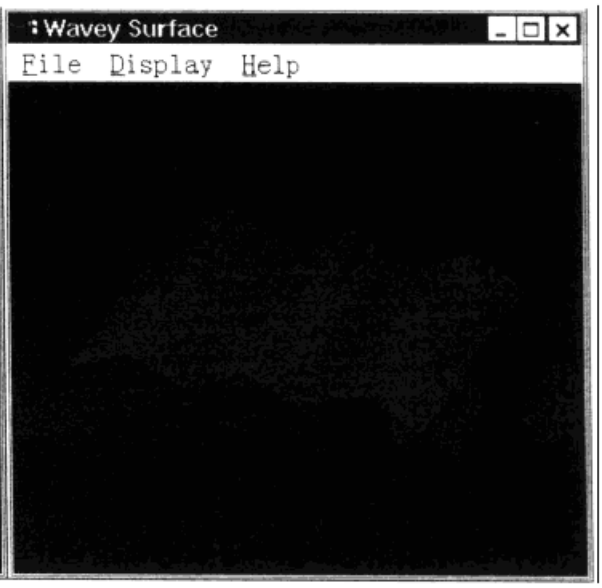


图 8-10 有法线平均的图形

程序 8-5 的绘图函数创建如图 8-8 所示的物面。比较图 8-9 没有法线平均操作与图 8-10 有法线平均操作的图形，可以看出，图 8-10 的图像上颜色变化比较光滑。

程序 8-5

```
// Called to draw scene
void RenderScene(void)
    {
    float normal[3];    // Storage for calculate normal
    float v[4][3];      // Storage for rectangle coordinates
    float lastY;        // Left hand side of rectangle
    float nextY;        // Right hand side of rectangle
    float temp;             // Temporary storeage for swapping
    float x;            // X coordinate storeage

    // Menu state specifies if wire frame or not
    if(iState == WIRE)
        glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);

    // Menu state specifies if smooth or flat shading
    if(iState == SMOOTH || iState == AVERAGE)
        glShadeModel(GL_SMOOTH);
    else
        glShadeModel(GL_FLAT);

    // Clear the window with current clearing color
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Save the matrix state and do the rotations
glPushMatrix();
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Set surface color to blue
glRGB(0,0,255);

// Initialize the Y steppings
lastY = 0.0f;
nextY = 10.0f;

// Loop through x coordinate from left to right, build
// a rectangle with alternating slops upwards and downwards
for(x = -60.0f; x < 60.0f; x+= 20.0f)
     {
     // 1st Vertices
     v[0][0] = x;        // X coord for left
     v[0][1] = lastY;
     v[0][2] = 50.0f;    // Z coord for back

     // 2nd vertices
     v[1][0] = x;        // X coord for left
     v[1][1] = lastY;
     v[1][2] = -50.0f;   // Z coord for front

     // 3rd Vertices
     v[2][0] = x + 20.0f; // X coord for right
     v[2][1] = nextY;
     v[2][2] = -50.0f;   // Z coord for front

     // 4rth Vertices
     v[3][0] = x + 20.0f;// X coord for right
     v[3][1] = nextY;
     v[3][2] = 50.0f;    // Z coord for back

     // Begin the polygon
     glBegin(GL_POLYGON);
          if(iState != AVERAGE)
```

```
        {
        // Calculate and set the normal vector, unless averaging
        // selected from the menu.
        calcNormal(v,normal);
        glNormal3fv(normal);
        }
    else // Average normals. Here we cheat because we know the normal
            // Points either up or down
        {
        // Normal points straight up
        if(nextY == 10)
            glNormal3f(0.0f,1.0f, 0.0f);
        else
            // Normal points straight down
            glNormal3f(0.0f,-1.0f, 0.0f);
        }

    // Specify the left two verticies
    glVertex3fv(v[0]);
    glVertex3fv(v[1]);

    // Do the same, the normal on the other side points the other
    // direction though
    if(iState == AVERAGE)
        {
        if(nextY == 10)
            glNormal3f(0.0f,-1.0f, 0.0f);// Normal points down
        else
            glNormal3f(0.0f,1.0f, 0.0f); // Normal points up
        }

    // Specify the right two verticies
    glVertex3fv(v[2]);
    glVertex3fv(v[3]);
glEnd();

// Swap the Y coordinate positions
temp = lastY;
lastY = nextY;
nextY = temp;
}
```

```
    glPopMatrix();

    // Flush drawing commands
    glFlush();
    }
```

⑥聚光灯

创建一个聚光灯(点光源)的过程与其他有方向光源一样，程序 8-6 在窗口中心放置一个蓝色球体，用上、下向键可以垂直移动一个聚光灯（用一个红色圆锥指示聚光灯的位置，在其内部放置一个黄色圆球代表一个灯泡），左、右键水平移动这个聚光灯。移动过程中，在球体表面出现一个镜面高光(如图 8-11 所示)。

程序 8-6

```
// Light values and coordinates
GLfloat    lightPos[] = { 0.0f, 0.0f, 75.0f, 1.0f };
GLfloat  specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
GLfloat  specref[] =  { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat  ambientLight[] = { 0.5f, 0.5f, 0.5f, 1.0f};
GLfloat  spotDir[] = { 0.0f, 0.0f, -1.0f };

// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
    {
    glEnable(GL_DEPTH_TEST); // Hidden surface removal
    glFrontFace(GL_CCW);          // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);       // Do not try to display the back sides

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Setup and enable light 0
    // Supply a slight ambient light so the objects can be seen
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLight);

    // The light is composed of just a diffuse and specular components
    glLightfv(GL_LIGHT0,GL_DIFFUSE,ambientLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);

    // Specific spot effects
```

```
    // Cut off angle is 60 degrees
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);

    // Fairly shiny spot
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 100.0f);

    // Enable this light in particular
    glEnable(GL_LIGHT0);

    // Enable color tracking
    glEnable(GL_COLOR_MATERIAL);

    // Set Material properties to follow glColor values
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // All materials hereafter have full specular reflectivity
    // with a high shine
    glMaterialfv(GL_FRONT, GL_SPECULAR, specref);
    glMateriali(GL_FRONT, GL_SHININESS, 128);

    // Black background
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    }

// Called to draw scene
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Set material color and draw a sphere in the middle
    glRGB(0, 0, 255);
    auxSolidSphere(30.0f);

    // Now place the light
    // Save the coordinate transformation
    glPushMatrix();
        // Rotate coordinate system
        glRotatef(yRot, 0.0f, 1.0f, 0.0f);
        glRotatef(xRot, 1.0f, 0.0f, 0.0f);
```

```
        // Specify new position and direction in rotated coords.
        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
        glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotDir);

        // Draw a red cone to enclose the light source
        glRGB(255, 0, 0);

        // Translate origin to move the cone out to where the light
        // is positioned.
        glTranslatef(lightPos[0], lightPos[1], lightPos[2]);
        auxSolidCone(4.0f, 6.0f);

        // Draw a smaller displaced sphere to denote the light bulb
        // Save the lighting state variables
        glPushAttrib(GL_LIGHTING_BIT);

        // Turn off lighting and specify a bright yellow sphere
            glDisable(GL_LIGHTING);
            glRGB(255, 255, 0);
            auxSolidSphere(3.0f);

        // Restore lighting state variables
        glPopAttrib();

    // Restore coordinate transformations
    glPopMatrix();

    // Flush drawing commands
    glFlush();
    }
```

下面的程序段定义一个聚光灯，其截止角为 60° 半锥角，在这一锥形范围外没有光线照亮，如图 8-12 所示 GL_SPOT_CUTOFF 定义截止角度。

```
    // Specific spot effects
    // Cut off angle is 60 degrees
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 60.0f);

    // Fairly shiny spot
    glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 100.0f);
```

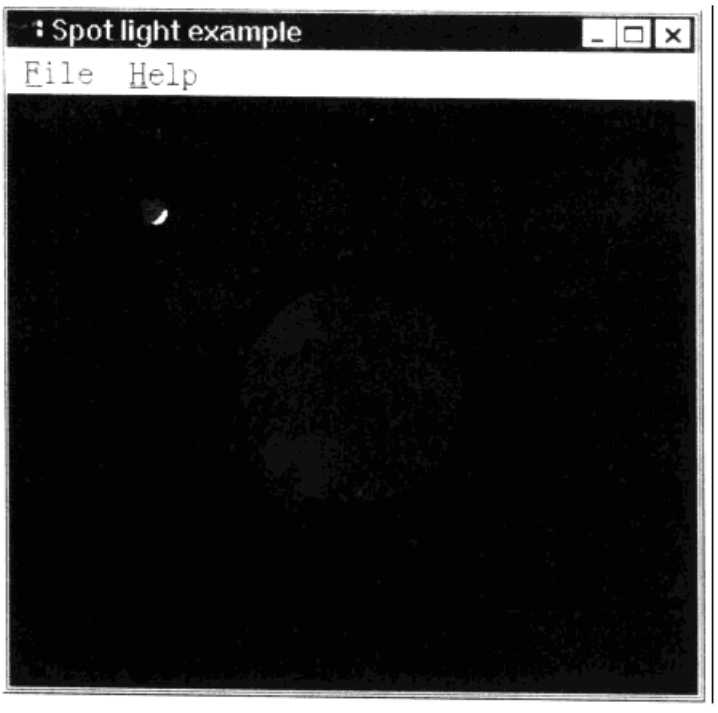


图 8-11　使用聚光灯

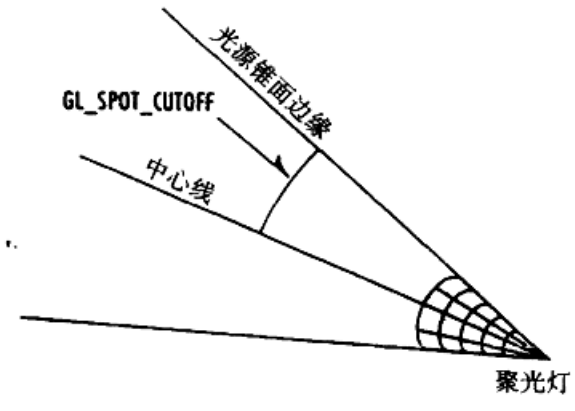


图 8-12　GL_SPOT_CUTOFF 的示意图

8.4　阴影处理

只要谈到光照处理，就不能不讲述阴影处理的技巧。在场景中加上阴影可以极大地提高图像的逼真程度和视觉效果，但是 OpenGL 并没有专门用于处理阴影的函数。在场景中加入阴影效果需要一些技巧，本节就将讲述这些技巧，并给出具体的实现步骤。

1．什么是阴影

阴影是由一个物体遮挡住其后的物体或表面，使发自光源的光线不能照亮这些部分造成的。在编程时，可以把物体投影在所在的平面上，并把这个投影绘成黑色或其他暗颜色，有时要做半透明处理(参见本书第十二章的例子)，如图 8-13 所示，实现上述过程要用到本书第六章讲述的矩阵操作技巧。

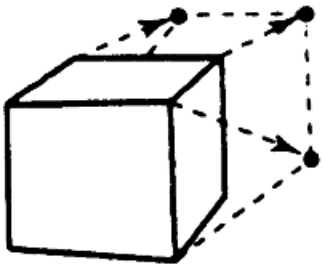


图 8-13　创建一个阴影

2. 压扁程序

(1) 要在场景中创建一个阴影，就要把模型、取景和投影矩阵“压扁”，使其描述的物体处在一个二维平面上，而压扁后在平面上的位置就是阴影所在的位置。

(2) 另外要考虑光源的距离和方向，光线的方向决定阴影的形状、大小。下面的程序段（程序 8-7）用于在准备设置阴影的平面上取三点，程序段中列出了导出阴影所在平面的程序系数，并同时利用光源位置构造一个变化矩阵，函数最终将构造一个指向变换矩阵的指针。

程序 8-7

```
// Creates a shadow projection matrix out of the plane equation
// coefficients and the position of the light. The return value is stored
// in destMat[][]
void MakeShadowMatrix(GLfloat points[3][3], GLfloat lightPos[4], GLfloat destMat[4][4])
    {
    GLfloat planeCoeff[4];
    GLfloat dot;

    // Find the plane equation coefficients
    // Find the first three coefficients the same way we
    // find a normal.
    calcNormal(points,planeCoeff);

    // Find the last coefficient by back substitutions
    planeCoeff[3] = - (
        (planeCoeff[0]*points[2][0]) + (planeCoeff[1]*points[2][1]) +
        (planeCoeff[2]*points[2][2]));

    // Dot product of plane and light position
    dot = planeCoeff[0] * lightPos[0] +
          planeCoeff[1] * lightPos[1] +
          planeCoeff[2] * lightPos[2] +
          planeCoeff[3] * lightPos[3];
```

```
    // Now do the projection
    // First column
    destMat[0][0] = dot - lightPos[0] * planeCoeff[0];
    destMat[1][0] = 0.0f - lightPos[0] * planeCoeff[1];
    destMat[2][0] = 0.0f - lightPos[0] * planeCoeff[2];
    destMat[3][0] = 0.0f - lightPos[0] * planeCoeff[3];

    // Second column
    destMat[0][1] = 0.0f - lightPos[1] * planeCoeff[0];
    destMat[1][1] = dot - lightPos[1] * planeCoeff[1];
    destMat[2][1] = 0.0f - lightPos[1] * planeCoeff[2];
    destMat[3][1] = 0.0f - lightPos[1] * planeCoeff[3];

    // Third Column
    destMat[0][2] = 0.0f - lightPos[2] * planeCoeff[0];
    destMat[1][2] = 0.0f - lightPos[2] * planeCoeff[1];
    destMat[2][2] = dot - lightPos[2] * planeCoeff[2];
    destMat[3][2] = 0.0f - lightPos[2] * planeCoeff[3];

    // Fourth Column
    destMat[0][3] = 0.0f - lightPos[3] * planeCoeff[0];
    destMat[1][3] = 0.0f - lightPos[3] * planeCoeff[1];
    destMat[2][3] = 0.0f - lightPos[3] * planeCoeff[2];
    destMat[3][3] = dot - lightPos[3] * planeCoeff[3];
    }

// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat nRange = 200.0f;
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
```

```
    glLoadIdentity();

    fAspect = (GLfloat)w/(GLfloat)h;

    gluPerspective(60.0f, fAspect, 1.0, 500.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // Move out Z axis so we can see everything
    glTranslatef(0.0f, 0.0f, -400.0f);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    }
```

如果用这一变换矩阵与当前模型取景矩阵相乘，则以后绘制的图就出现在这个平面上。

3. 阴影的例子

作为上面函数的应用实例，现在把飞机放在空中，并在其左上方放置一个光源。可以用方向键旋转飞机，显示的结果见图 8-14。

下面的程序为本例中创建阴影投影矩阵的程序段，该矩阵在创建后，存贮成一个全局变量。

```
// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
    {
    // Any three points on the ground (counter clockwise order)
    GLfloat points[3][3] = {{ -30.0f, -149.0f, -20.0f },
                            { -30.0f, -149.0f, 20.0f },
                            { 40.0f, -149.0f, 20.0f }};

    glEnable(GL_DEPTH_TEST); // Hidden surface removal
    glFrontFace(GL_CCW);         // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);      // Do not calculate inside of jet

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Setup and enable light 0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
```

```
    glEnable(GL_LIGHT0);

    // Enable color tracking
    glEnable(GL_COLOR_MATERIAL);

    // Set Material properties to follow glColor values
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    // All materials hereafter have full specular reflectivity
    // with a high shine
    glMaterialfv(GL_FRONT, GL_SPECULAR,specref);
    glMateriali(GL_FRONT,GL_SHININESS,128);

    // Light blue background
    glClearColor(0.0f, 0.0f, 1.0f, 1.0f );

    // Calculate projection matrix to draw shadow on the ground
    MakeShadowMatrix(points, lightPos, shadowMat);
    }
```

下面为绘制程序段。首先，画飞机，再恢复模型取景矩阵，使之与投影矩阵相乘，得到“压扁”投影矩阵，然后再绘制飞机；再次恢复模型取景矩阵，在光源处放置一个黄色小球，在与阴影同一平面上画一个矩形。

```
// Called to draw scene
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glPushMatrix();

    // Draw jet at new orientation, put light in correct position
    // before rotating the jet
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glRotatef(xRot, 1.0f, 0.0f, 0.0f);
    glRotatef(yRot, 0.0f, 1.0f, 0.0f);

    DrawJet(FALSE);

    // Restore original matrix state
    glPopMatrix();
```

```
// Get ready to draw the shadow and the ground
// First disable lighting and save the projection state
glPushAttrib(GL_LIGHTING_BIT);
glDisable(GL_LIGHTING);
glPushMatrix();

// Multiply by shadow projection matrix
glMultMatrixf((GLfloat *)shadowMat);

// Now rotate the jet around in the new flattend space
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);

// Pass true to indicate drawing shadow
DrawJet(TRUE);

// Restore the projection to normal
glPopMatrix();

// Draw the light source
glPushMatrix();
glTranslatef(lightPos[0],lightPos[1], lightPos[2]);
glRGB(255,255,0);
auxSolidSphere(5.0f);
glPopMatrix();

// Draw the ground, we do manual shading to a darker green
// in the background to give the illusion of depth
glBegin(GL_QUADS);
    glRGB(0,128,0);
    glVertex3f(400.0f, -150.0f, -200.0f);
    glVertex3f(-400.0f, -150.0f, -200.0f);
    glRGB(0,255,0);
    glVertex3f(-400.0f, -150.0f, 200.0f);
    glVertex3f(400.0f, -150.0f, 200.0f);
glEnd();

// Restore lighting state variables
glPopAttrib();
```

```
// Flush drawing commands
glFlush();
}
```

图 8-14　阴影处理的例子

8.5　光照处理与色彩指数模式

在第七章中提到色彩指数模式用颜色的指数，而不是 RGB 三组分定义一个颜色，而光照处理时，定义光源的材质属性都要用到 RGBA 组分。

在 OpenGL 的色彩指数模式下，光照属性只能在漫反射和镜面成分材质属性中可以有光洁度、泛光、漫反射、镜面反射属性。

在做光照处理时，调色板中要为泛光、漫反射、镜面属性建立三个颜色序列。为得到满意的效果，这些序列的颜色要由黑到某一单一颜色再到白色排列。

通常建议不要在色彩指数模式下做光照处理。程序 8-8 为在色彩指数模式下做光照处理的例子。

程序 8-8

```
// Light values and coordinates
GLfloat    lightPos[] = { -50.0f, 100.0f, -150.0f, 0.0f };

// Ambient, diffuse, and specular indexes into palette
GLfloat    matColorMap[] = { 16.0f, 47.0f, 79.0f };
GLfloat    matShine[] = { 120.0f };
```

```
// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
    {
    glEnable(GL_DEPTH_TEST); // Hidden surface removal
    glFrontFace(GL_CW);      // Counter clock-wise polygons face out
    glEnable(GL_CULL_FACE);      // Do not try to display the back sides
    glShadeModel(GL_SMOOTH); // Smooth shading

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Enable this light in particular
    glEnable(GL_LIGHT0);

    // Set color ramp values for material colors
    glMaterialfv(GL_FRONT, GL_COLOR_INDEXES, matColorMap);
    glMaterialfv(GL_FRONT, GL_SHININESS, matShine);

    // Clear with first color in Blue color ramp. In this case, Black
    glClearIndex(16.0f);
    }

// Called to draw scene
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glPushMatrix();
        glTranslatef(-50.0f, 0.0f, 0.0f);
        auxSolidSphere(25.0f);
        glTranslatef(50.0f, 0.0f, 0.0f);

        glPushMatrix();
            glRotatef(45.0f, 1.0f, 1.0f, 0.0f);
            auxSolidCube(25.0f);
        glPopMatrix();

        glTranslatef(50.0f, -15.0f, 0.0f);
```

```
    glRotatef(280.0f, 1.0f, 0.0f, 0.0f);
    auxSolidCone(25.0f, 35.0f);
glPopMatrix();

// Flush drawing commands
glFlush();
}
```

8.6 相关函数

```
void glColorMaterial(GLenum face, GLenum mode);
void glCullFace(GLenum mode);
void glFrontFace (GLenum mode);
void glGetMaterial {fi}v (GLenum face, GLenum pname, TYPE *params);
void glGetLightfv (GLenum light, GLenum pname, GLfloat *params);
void glGetLightiv (GLenum light, GLenum pname, GLint *params);
void glLight {fi} (GLenum light, GLenum pname, TYPE param);
void glLightModel {fi} (GLenum pname, TYPE param);
void glMaterial1{fi} (GLenum face, GLenum pname, TYPE param);
void glMaterial1{fi}v(GLenum face, GLenum pname, const TYPE *params);
void glNormal3{bdfis} (TYPE nx, TYPE ny, TYPE nz);
void glNormal3{bdfis}v (const TYPE *v);
```

第九章　在 OpenGL 中使用光栅图形

光栅图形是由彩色的二维数组构成的，不仅可以用于在屏幕上显示三维图形，也用于在光栅打印机(或动画片)上打印图像。OpenGL 提供了直接管理二维位图和图像的函数，本章将讲述这些函数的使用。

9.1　绘制位图

OpenGL 的位图是用于在屏幕上快速绘制字符或符号的两色图像，与 Microsoft 定义有多个颜色的图像不同。OpenGL 绘制位图的函数是 glBitmap。用 glBitmap 绘图时，0 颜色表示透明，1 表示用当前颜色和光照材质属性画图。

程序 9-1 所示为一个绘制位图的程序，包括一个位图数据和 glBitmap 调用，所显示的图像见图 9-1。

程序 9-1

```
void RepaintWindow(RECT *rect)     /* I - Client area rectangle */
{
  int          i;         /* Looping var */
  static GLubyte smiley[] =   /* 16x16 smiley face */
  {
    0x03, 0xc0, 0, 0, /*       ****       */
    0x0f, 0xf0, 0, 0, /*     ********     */
    0x1e, 0x78, 0, 0, /*    ****  ****    */
    0x39, 0x9c, 0, 0, /*   ***  **  ***   */
    0x77, 0xee, 0, 0, /*  *** ****** ***  */
    0x6f, 0xf6, 0, 0, /*  ** ******** **  */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0x73, 0xce, 0, 0, /*  ***  ****  ***  */
    0x73, 0xce, 0, 0, /*  ***  ****  ***  */
    0x3f, 0xfc, 0, 0, /*   ************   */
    0x1f, 0xf8, 0, 0, /*    **********    */
    0x0f, 0xf0, 0, 0, /*     ********     */
    0x03, 0xc0, 0, 0  /*       ****       */
  };
```

```
    glViewport(0, 0, rect->right, rect->bottom);

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, rect->right - 1.0, 0.0, rect->bottom - 1.0, -1.0, 1.0);

  /*
   * This bitmap is aligned to 4-byte boundaries...
   */

    glPixelTransferi(GL_UNPACK_ALIGNMENT, 4);

    glColor3f(1.0, 0.0, 0.0);
    for (i = 0; i < 100; i ++)
    {
      glRasterPos2i(rand() % rect->right, rand() % rect->bottom);
      glBitmap(16, 16, 8.0, 8.0, 0.0, 0.0, smiley);
    };

    glFinish();
}
```

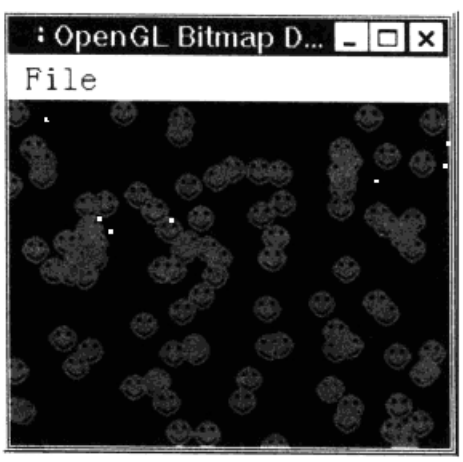


图 9-1　绘制位图

在程序中定义了一个 16×16 像素的位图。OpenGL 的位图是按由下至上顺序存贮的，如果要定义成由上至下形式的位图，则要指定一个负高度，同时，Microsoft OpenGL 库函数要求位图的每行以 4 个字节的边界对齐，可以用 glPixelStore 函数改变位图的对齐量。

定义了位图后，还要调用 glRasterPos 函数指定当前光栅坐标：

```
glRasterPos2i(rand() % rect->right, rand() % rect->bottom);
```

上面的程序在窗口绘图区的随机位置绘制位图。调用 glRasterPos 时，该函数检验光栅位置是否有效，如果有效，把光栅位置有效标识置为 True，否则为 False。

值得注意的是，多边形和其他矢量绘图的几何要素即使有一部分在当前视区外，也会经过裁剪，在视见区中有另一部分可见，而绘制位图时，如果光栅位置不在视见区内，则根本不会绘制该位图。

在绘制位图时，调用 glBitmap 函数

```
glBitmap(16, 16, 8.0, 8.0, 0.0, 0.0, smiley);
```

把一个 16×16 位图的位图中心放置在(8.0,8.0)处，绘制位图后，光栅位置移至(0.0，0.0)。该函数的定义如下：

```
void glBitmap(GLsizei width, GLsizei height, GLfloat xorig, GLfloat yorig,
              GLfloat xmove, GLfloat ymove, const GLubyte *bits);
```

width，height 分别为位图的宽、高，xorig，yorig 分别为位图中心的 x,y 坐标，绘制位图后，光栅位置移到(xmove,ymove)处，不改变光栅位置有效标识。xomve，ymove 参数通常用于绘制位图字体时，把光栅坐标放在下一个字符处。

1．位图形式的字体

位图的一个重要应用是在屏幕上显示字符串。用户可以为每一个字符定义一个位图矩阵，在显示该字符时，在屏幕上绘制相应的位图。Microsoft Win32 库提供了一个 wglUseFontBitmaps 函数可以利用系统的字体文件生成这些位图。

使用这些字体位图时，要用到 OpenGL 的 glGenLists,glListBase，glCallList 三个函数。glGenLists 函数生成一个连续的由 wglUseFontBitmaps 函数生成的有关字体位图 OpenGL 显示列表。

```
GLuint base;
HDC hdc;

base=glGenLists(96);
wglUseFontBitmaps(hdc, 32, 96, base);
```

由当前字体的第 32 个字符(即 ASCII 码表的空格)开始，创建 96 个字符位图。base 变量是第一个字体位图显示列表。下面的程序段用这些位图显示字符串。

```
char *s;

glListBase(base-32);
glCallLists(strlen(s), GL_UNSIGNED_BYTE, s);
```

2．创建一个简单的字体库

当然，使用 wglCreateFontBitmaps 函数可以简化字体的创建过程，但是最终输出字符串还要有很多过程。如果用户自己创建一个字体库，也是相当方便的。程序 9-2 是创建字体的函数。

程序 9-2

```
GLuint FontCreateBitmaps(HDC   hdc,          /* I - Device Context */
                         char  *typeface, /* I - Font specification */
                         int   height,        /* I - Font height/size in pixels */
```

```
            int   weight,       /* I - Weight of font (bold, etc) */
            DWORD italic)       /* I - Text is italic */
{
  GLuint  base;               /* Base display list for font */
  HFONT       font;               /* Windows font ID */

  if ((base = glGenLists(96)) == 0)
    return (0);
```

typeface 参量是字体名称，height, weight, italic 参量直接传递给 wglUseFontBitmaps 函数，用于设置字符的尺寸和形状。在创建字体位图前，要先选定一个字符集，通常选用 ANSI 或 UNICODE 字符集。ANSI 字符集(ANSI_CHARSET)为标准 7 位 ASCII 字符集，UNICODE 字符集(UNICODE_CHARSET)支持国际字符和区位字符集。有些字体使用特殊字符，符号字体可提供希腊字和科学符号。

程序 9-2 使用 ANSI_CHARSET 作为常规字体，SYMBOL_CHARSET 为符号字体。如下面的程序段所示。

```
  if (stricmp(typeface, "symbol") == 0)
    font = CreateFont(height, 0, 0, 0, weight, italic, FALSE, FALSE,
                      SYMBOL_CHARSET, OUT_TT_PRECIS,
                      CLIP_DEFAULT_PRECIS, DRAFT_QUALITY,
                      DEFAULT_PITCH  typeface);
  else
    font = CreateFont(height, 0, 0, 0, weight, italic, FALSE, FALSE,
                      ANSI_CHARSET, OUT_TT_PRECIS,
                      CLIP_DEFAULT_PRECIS, DRAFT_QUALITY,
                      DEFAULT_PITCH, typeface);

  SelectObject(hdc, font);

  wglUseFontBitmaps(hdc, 32, 96, base);

  return (base);
}
```

下面的程序段为字体删除函数，删除指定的显示列表。如果使用国际字符集，要把显示列表数目由 96 改为 224。

```
void
FontDelete(GLuint font)  /* I - Font to delete */
{
  if (font == 0)
    return;

  glDeleteLists(font, 96);
```

```
}
```

注意：

(1)要用 glPushAttrib 和 glPopAttrib 函数存贮、恢复以 ID 开始的显示列表，否则会影响要使用显示列表的其他程序。

(2)上面提到的字符串中不能有 Tab 和换行符，否则会影响最终输出结果，在使用 glCallList 函数前，要对输入字符串进行分析。

```
void
FontPuts(GLuint font,    /* I - Font to use */
         char   *s) /* I - String to display */
{

  if (font == 0)
    return;

  if (s == NULL)
    return;

  glPushAttrib(GL_LIST_BIT);
    glListBase(font - 32);
    glCallLists(strlen(s), GL_UNSIGNED_BYTE, s);
  glPopAttrib();
}
```

9.2 Pixmaps——彩色位图

彩色位图通常称为 pixmaps(为 pixel maps 的缩写)。与 glBitmap 相似，glDrawpixels 函数把图像左下角定义在当前光栅位置，但不能指定光栅原点和位移量。

```
BITMAPINFO *BitmapInfo;
GLubyte *BitmapBits;

glRastePos2I(xoffset, yoffset);
glDrawPixels(BitmapInfo->bmiHeader.biWidth,
             BitmapInfo->bmiHeader.biHeight,
                GL_RGB, GL_UNSIGNED, BitmapBits);
```

glDrawPixels 函数有五个参量：

```
glDrawPixels(GLsizei width, GLsizei height, GLenum format,
                GLenum type, GLvoid *pixels);
```

format 参数指定 pixmap 的色彩空间，有效的值是 GL_COLOR_INDEX，GL_RGB 和 GL_LUMINANCE 之一。GL_COLOR_INDEX 指 pixmap 的颜色定义为当前窗口逻辑调色板的一个指数，常用于图标；GL_LUMINANCE 把每一个颜色映射成屏幕上的灰度值，最小值为黑色，

最大值为白色；GL_RGB 格式把图像的每一个像素指定为一个 RGB 值。

type 参数的类型和每一颜色或组分的取值范围，有效值为 GL_BYTE(取值范围[-128,127])， GL_UNSIGNED_BYTE(取值范围为[0,255])和 GL_BITMAP(取值范围为 0,1)。

1．重新映射颜色

当 format 参数选用 GL_COLOR_INDEX 时，可以调用 glPixelMap 或 glPixelTransfer 函数把 pixmap 或位图的颜色重新映射。glPixelTransfer 函数可以指定色彩指数和 RGB 值的缩放、偏置值，如把 RGB 图像亮度加大 10%：

```
glPixelTransferf(GL_RED_SCALE, 1.1);
glPixelTransferf(GL_GREEN_SCALE, 1.1);
glPixelTransferf(GL_BLUE_SCALE, 1.1);
```

同样，把位图的色彩指数偏移到所定义调色板的条目：

```
glPixelTransferi(GL_INDEX_OFFSET, bitmap_entry);
```

下面的程序段把位图的两个颜色重新映射成不同指数：

```
void
RepaintWindow(RECT *rect)     /* I - Client area rectangle */

{
  int          i;         /* Looping var */
  static GLubyte smiley[] =  /* 16x16 smiley face */
  {
    0x03, 0xc0, 0, 0, /*       ****       */
    0x0f, 0xf0, 0, 0, /*     ********     */
    0x1e, 0x78, 0, 0, /*    ****  ****    */
    0x39, 0x9c, 0, 0, /*   ***  **  ***   */
    0x77, 0xee, 0, 0, /*  *** ****** ***  */
    0x6f, 0xf6, 0, 0, /*  ** ******** **  */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0xff, 0xff, 0, 0, /* **************** */
    0x73, 0xce, 0, 0, /*  ***  ****  ***  */
    0x73, 0xce, 0, 0, /*  ***  ****  ***  */
    0x3f, 0xfc, 0, 0, /*   ************   */
    0x1f, 0xf8, 0, 0, /*    **********    */
    0x0f, 0xf0, 0, 0, /*     ********     */
    0x03, 0xc0, 0, 0  /*       ****       */
  };

  glViewport(0, 0, rect->right, rect->bottom);

  glClearIndex(0.0);
```

```
 glClear(GL_COLOR_BUFFER_BIT);

 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 glOrtho(0.0, rect->right - 1.0, 0.0, rect->bottom - 1.0, -1.0, 1.0);

/*
 * This bitmap is aligned to 4-byte boundaries...
 */

 glPixelTransferi(GL_UNPACK_ALIGNMENT, 4);
 glPixelTransferi(GL_INDEX_OFFSET, 1);

 for (i = 0; i < 100; i ++)
 {
   glRasterPos2i(rand() % rect->right, rand() % rect->bottom);
   glDrawPixels(16, 16, GL_COLOR_INDEX, GL_BITMAP, smiley);
 };

 glFinish();
}
```

2. 颜色映射表

有时，有必要做一些比线性缩放和偏置更复杂的色彩校正，如 gama 校正可以把每一个颜色值调整成一个指数曲线分布的值以补偿显示器或打印机上出现的不均匀性。

可以调用 glDixelMap 定义一个查对表，完成这一校正操作。

```
GLfloat lut[256];
GLfloat gama_value;
int i;

gama_value=1.7; /* for NTSC video monitors */
for (i=0; i<256; i++)
     lut[i]=pow(i/255.0, 1.0/gamma_value);

glPixelTransteri(GL_MAP_COLOR_, GL_TRUE);
glPixelMap(GL_PIXEL_MAP_R_TO_R, 256, lut);
glPixelMap(GL_PIXEL_MAP_G_TO_G, 256, lut);
glPixelMap(GL_PIXEL_MAP_B_TO_B, 256, lut);
```

3. 缩放一个 pixmap

除可以调整 pixmap 的色彩外，还可以用 glPixelZoom 函数调整 pixmap 的尺寸。该函数有两个浮点参数，分别定义图像 X、Y 方向的缩放因子。

```
glPixelZoom(1.0, 1.0);  /* Don't scale the image */
glPixelZoom(-1.0, 1.0); /* Flip the image horizontally */
glPixelZoom(1.0, -2.0); /* Flip the image and double the height */
glPixelZoom(0.33, 0.33); /* Draw the image 1/3 size */
```

如果缩放因子为负值，把图像在该方向翻转过来。其他非线性的效果，需要用到纹理映射(参见本书第十章)。

glPixelStore 函数可以移动观察点，观察图像的不同部分，如显示 640×480 图像的 300×300 中心区域。

```
glPixelStorei(GL_UNPACK_ROW_LENGTH, 640);
glPixelStorei(GL_UNPACK_SKIP_PIXELS, (640-300)/2);
glPixelStorei(GL_UNPACK_SKIP_ROW, (480-300)/2);
glDrawPixels(300, 300, GL_RGB, GL_UNSIGNED_BYTE, BitmapBits);
```

GL_UNPACK_ROW_LENGTH 指定原图像的宽度，在 glDrawPixels 指定的宽度与图像宽度不同时使用该参数。

GL_UNPACK_SKIP_PIXELS 指定从图像左边跳过的像素数目。本例中跳过(640-300)/2。

GL_UNPACK_SKIP_ROWS 指定跳过图像的行数。通常为从底部开始，但可以用 glPixelZoom 指定一个负 Y 值缩放因子加以改变。

注意：上面提到的属性是指以像素为单位的 pixmap 尺寸，而不是缩放后的尺寸。

4. 读入 pixmaps

OpenGL 的 glReadPixels 函数可以从屏幕上读入一个图像。除了把它创建的图像存到磁盘上，也可以用于纹理映射产生逼真效果。

与 glDrawPixels 不同，glReadPixels 函数忽略当前光栅坐标而要用户指定一个要读入图像的左下角视见区坐标(x,y)。程序 9-3 演示如何在当前视见区读入一个可以存成一个文件或作为纹理的 Windows 位图结构。

程序 9-3

```
/*
 * 'ReadDIBitmap()' - Read the current OpenGL viewport into a
 *                    24-bit RGB bitmap.
 *
 * Returns the bitmap pixels if successful and NULL otherwise.
 */

void *
ReadDIBitmap(BITMAPINFO **info)        /* O - Bitmap information */
{
  long        i, j,                    /* Looping var */
              bitsize,       /* Total size of bitmap */
              width;               /* Aligned width of a scanline */
  GLint       viewport[4];             /* Current viewport */
  void        *bits;                   /* RGB bits */
```

```
GLubyte *rgb,                /* RGB looping var */
        temp;                /* Temporary var for swapping */

/*
 * Grab the current viewport...
 */

 glGetIntegerv(GL_VIEWPORT, viewport);

/*
 * Allocate memory for the header and bitmap...
 */

 if ((*info = (BITMAPINFO *)malloc(sizeof(BITMAPINFOHEADER))) == NULL)
 {
  /*
   * Couldn't allocate memory for bitmap info - return NULL...
   */

   return (NULL);
 };

 width   = viewport[2] * 3;       /* Real width of scanline */
 width   = (width + 3) & ~3;      /* Aligned to 4 bytes */
 bitsize = width * viewport[3];   /* Size of bitmap, aligned */

 if ((bits = calloc(bitsize, 1)) == NULL)
 {
  /*
   * Couldn't allocate memory for bitmap pixels - return NULL...
   */

   free(*info);
   return (NULL);
 };

 /*
  * Read pixels from the framebuffer...
  */

  glFinish();                 /* Finish all OpenGL commands */
```

```
  glPixelStorei(GL_PACK_ALIGNMENT, 4); /* Force 4-byte alignment */
  glPixelStorei(GL_PACK_ROW_LENGTH, 0);
  glPixelStorei(GL_PACK_SKIP_ROWS, 0);
  glPixelStorei(GL_PACK_SKIP_PIXELS, 0);

  glReadPixels(0, 0, viewport[2], viewport[3], GL_RGB, GL_UNSIGNED_BYTE,
               bits);

 /*
  * Swap red and blue for the bitmap...
  */

  for (i = 0; i < viewport[3]; i ++)
    for (j = 0, rgb = ((GLubyte *)bits) + i * width;
         j < viewport[2];
      j ++, rgb += 3)
    {
      temp   = rgb[0];
      rgb[0] = rgb[2];
      rgb[2] = temp;
    };

 /*
  * Finally, initialize the bitmap header information...
  */

  (*info)->bmiHeader.biSize          = sizeof(BITMAPINFOHEADER);
  (*info)->bmiHeader.biWidth         = viewport[2];
  (*info)->bmiHeader.biHeight        = viewport[3];
  (*info)->bmiHeader.biPlanes        = 1;
  (*info)->bmiHeader.biBitCount      = 24;
  (*info)->bmiHeader.biCompression   = BI_RGB;
  (*info)->bmiHeader.biSizeImage     = bitsize;
  (*info)->bmiHeader.biXPelsPerMeter = 2952; /* 75 DPI */
  (*info)->bmiHeader.biYPelsPerMeter = 2952; /* 75 DPI */
  (*info)->bmiHeader.biClrUsed       = 0;
  (*info)->bmiHeader.biClrImportant  = 0;

  return (bits);
}
```

首先，要调用 glGetIntegerv 函数得到当前视见区尺寸的信息(详细内容请参见《精通

OpenGL》一书)，把当前原点(x,y)，X,Y 尺寸存入 viewport 数组。

表 9-1 viewport 数组定义

viewport 数组	数 值
viewport[0]	视见区原点 x 坐标
viewport[1]	视见区原点 y 坐标
viewport[2]	视见区 x 方向尺寸
viewport[3]	视见区 y 方向尺寸

在得到视见区尺寸后，就要为 pixmap 分配内存。应该注意的是，Windows 位图(缺省情况下的 OpenGL Pixmaps)的每行应该在 32 位边界上开始。为此，

```
width  = viewport[2] * 3;      /* Real width of scanline */
width  = (width + 3) & ~3;     /* Aligned to 4 bytes */
```

把视见区宽度向上圆整到最近的 32 位边界值，Pixmap 的尺寸为：

```
bitsize = width * viewport[3];  /* Size of bitmap, aligned */
```

其后，调用 glReadPixels 函数获取当前视见区内容，并用必要的信息填充 Windows 的 BITMAPHEADER 结构

5. 拷贝 Pixmaps

OpenGL 提供了把屏幕上某一区域拷贝到另一位置的函数 glCopyPixels：

```
int mousex, mousey;

glReadBuffer(GL_FRONT);
glDrawBuffer(GL_FRONT);
glRasterPos2i(0, 0);
glCopyPixels(mousex-8, mousey-8, 16,16,GL_COLOR);
```

用 glCopyPixels 函数把像素从给定的位置拷贝到当前光栅位置：

```
glCopyPixels((GLint x, GLint y, GLsizei width, GLsizei height, GLenum type);
```

(x,y)为要拷贝区域左下角坐标，width，height 分别为所拷贝图像的宽、高，type 参量指定要拷贝的值。对大多数应用程序，像素类型为 GL_COLOR，即拷贝的像素类型为色彩指数或 RGB 值。

像素缩放操作是针对输出像素而不是输入像素的，例如，可以把 16×16 像素图像放大成 32×32 图像，glPixelStore 调用的偏置和尺寸不影响 glCopyPixels，而 glPixelTransfer 和 glPixelMap 的改变对其有影响。

9.3 位图浏览器

下面利用 OpenGL 提供的与位图相关函数，编写一个浏览 Windows BMP 文件的程序，这个程序要完成的功能有：

- 读入 Windows BMP 文件
- 把图像缩放到当前窗口尺寸
- 提供对图像亮度、gama 校正的简单控制
- 用鼠标指针给出一个图像放大视图

• 把显示的图像存盘

• 打印显示的图像

1. 关于 Windows 位图文件

Windows BMP 文件是广泛使用的一种图片格式。BMP 文件不采用数据压缩格式，便于在 OpenGL 程序中读、写等操作。

根据所使用颜色类型的不同，一个 BMP 文件由 3 或 4 部分组成。所有的 BMP 文件以 BITMAPFILEHEADER 结构开头，该结构有一个标识字符串（“BM”），文件大小，实际图像数据的偏移量，如图 9-2 所示。

RGB.BMP 文件	Palette.BMP 文件
BITMAPFILEHEADER	BITMAPFILEHEADER
BITMAPINFO	BITMAPINFO
位图或像素数据	PALETTEENTRY
	位图或像素数据

图 9-2 BMP 文件结构

BITMAPFILEHEADER 结构定义如下：

```
typedef struct tagBITMAPFILEHEADER
{
    WORD    bfType;        /* "BM" */
    WORD    bfSize;        /* Size of file in bytes */
    WORD    bfReserved1;/* Reserved, always 0 */
    WORD    bfReserved2; /* Reserved, always 0 */
    DWORD   bfoffBits;     /* Offset to image in bytes */
}   BITMAPFILEHEADER
```

其后为描述图像内容的结构，对于色彩指数图像，BITMAPINFOHEADER 结构的定义如下：

```
typedef struct tagBITMAPINFOHEADER{
   DWORD      biSize; /* Size of BITMAPINFOHEADER in bytes */
   LONG       biWidth; /* Width of image in pixels */
   LONG       biHeight; /* Height of image in pixels */
   WORD       biPlanes; /* No. of color planes (always 1)*/
   WORD       biBitCount; /* No. color bits */
   DWORD      biCompression; /* Type of compression used */
   DWORD      biSizeImage; /* Size of the image in bytes */
   LONG       biXPelsPerMeter; /* Horizintal pixels per meter */
   LONG       biYPelsPerMeter; /* Vertical pixels per meter */
   DWORD      biClrUsed; /* Number of color used */
   DWORD      biClrImportant; /* Number of important color */
} BITMAPINFOHEADER;
```

其后为图像中每一颜色的调色板，最后为图像数据。

2. 读入 BMP 文件

(1) 首先，打开文件并读入 BITMAPFILEHEADER 结构。

```
if ((fp = fopen(filename, "rb")) == NULL)
  return (NULL);

/*
 * Read the file header and any following bitmap information...
 */

if (fread(&header, sizeof(BITMAPFILEHEADER), 1, fp) < 1)
{
 /*
  * Couldn't read the file header - return NULL...
  */

  fclose(fp);
  return (NULL);
};
```

(2) 然后读入BITMAPINFO结构(和调色板定义)。

```
infosize = header.bfOffBits - sizeof(BITMAPFILEHEADER);
fread(*info, 1, infosize, fp)
```

(3) 最后读入位图数据并关闭文件。

```
if ((bitsize = (*info)->bmiHeader.biSizeImage) == 0)
  bitsize = ((*info)->bmiHeader.biWidth *
             (*info)->bmiHeader.biBitCount + 7) / 8 *
        abs((*info)->bmiHeader.biHeight);

  fread(bits, 1, bitsize, fp);
  fclose(fp);
```

3. 写BMP文件

与读入BMP文件的顺序相反，写入BMP文件时，只需把BITMAPFILEHEADER结构加入到内存中位图，再把内容写到磁盘上，即

```
int SaveDIBitmap(char       *filename, /* I - File to save to */
         BITMAPINFO *info,        /* I - Bitmap information */
            void       *bits)         /* I - Bitmap pixel bits */
{
  FILE             *fp;      /* Open file pointer */
  long             size,          /* Size of file */
              infosize, /* Size of bitmap info */
              bitsize;  /* Size of bitmap pixels */
  BITMAPFILEHEADER header;        /* File header */

 /*
```

```
* Try opening the file; use "wb" mode to write this *binary* file.
*/

if ((fp = fopen(filename, "wb")) == NULL)
  return (-1);

if (info->bmiHeader.biSizeImage == 0)/* Figure out the bitmap size */
  bitsize = (info->bmiHeader.biWidth *
             info->bmiHeader.biBitCount + 7) / 8 *
        abs(info->bmiHeader.biHeight);
else
  bitsize = info->bmiHeader.biSizeImage;

infosize = sizeof(BITMAPINFOHEADER);
switch (info->bmiHeader.biCompression)
{
  case BI_BITFIELDS :
      infosize += 12;            /* Add 3 RGB doubleword masks */
      if (info->bmiHeader.biClrUsed == 0)
     break;
  case BI_RGB :
      if (info->bmiHeader.biBitCount > 8 &&
          info->bmiHeader.biClrUsed == 0)
     break;
  case BI_RLE8 :
  case BI_RLE4 :
      if (info->bmiHeader.biClrUsed == 0)
        infosize += (1 << info->bmiHeader.biBitCount) * 4;
   else
        infosize += info->bmiHeader.biClrUsed * 4;
   break;
};

size = sizeof(BITMAPFILEHEADER) + infosize + bitsize;

/*
 * Write the file header, bitmap information, and bitmap pixel data...
 */

header.bfType      = 'MB';      /* Non-portable... sigh */
header.bfSize      = size;
```

```
header.bfReserved1 = 0;
header.bfReserved2 = 0;
header.bfOffBits   = sizeof(BITMAPFILEHEADER) + infosize;

if (fwrite(&header, 1, sizeof(BITMAPFILEHEADER), fp) < sizeof(BITMAPFILEHEADER))
{
 /*
  * Couldn't write the file header - return...
  */

  fclose(fp);
  return (-1);
};

if (fwrite(info, 1, infosize, fp) < infosize)
{
 /*
  * Couldn't write the bitmap header - return...
  */

  fclose(fp);
  return (-1);
};

if (fwrite(bits, 1, bitsize, fp) < bitsize)
{
 /*
  * Couldn't write the bitmap - return...
  */

  fclose(fp);
  return (-1);
};

/*
 * OK, everything went fine - return...
 */

 fclose(fp);
 return (0);
}
```

4. 打印位图

在下面的例子中，使用了标准 GDI 打印设备。因为 Windows 提供了几种在应用程序中打印的函数，下面的程序是与上面给出例子一起使用的。

(1) 首先，用 PrintDlg 显示标准 Windows 打印对话框。

```
memset(&pd, 0, sizeof(pd));
pd.lStructSize = sizeof(pd);
pd.hwndOwner   = owner;
pd.Flags       = PD_RETURNDC;
pd.hInstance   = NULL;
if (!PrintDlg(&pd))
  return (0);          /* User chose 'cancel'... */
```

若用户击 Cancel 按钮，返回 0 值，否则 PRINTDLG 结构返回一个用于打印的设备描述表 (HDC)。

(2) 下面，开始打印作业。

```
di.cbSize      = sizeof(DOCINFO);
di.lpszDocName = "OpenGL Image";
di.lpszOutput  = NULL;
StartDoc(pd.hDC, &di);
```

(3) 用 StretchBlt 函数绘制位图并结束打印作业。

```
StretchBlt(pd.hDC, xoffset, yoffset, xsize, ysize,
           hdc, 0, 0, info->bmiHeader.biWidth, info->bmiHeader.biHeight,
           SRCCOPY);
EndPage(pd.hDC);
EndDoc(pd.hDC);
```

(4) 计算基于输出页面尺寸的 StretchBlt 函数的前 4 个参数，把图像缩放到页面大小而长宽比保持不变。

```
xsize = rect.right;
ysize = xsize * info->bmiHeader.biHeight / info->bmiHeader.biWidth;
if (ysize > rect.bottom)
{
  ysize = rect.bottom;
  xsize = ysize * info->bmiHeader.biWidth / info->bmiHeader.biHeight;
};
```

X,Y 偏移量分别取宽、高方向差值的一半。

5. 显示位图

(1) 首先，显示位图的 OpenGL 程序设置当前视见区和取景变换。

```
glViewport(0, 0, rect->right, rect->bottom);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glOrtho(0.0, rect->right - 1.0, 0.0, rect->bottom - 1.0, -1.0, 1.0);
glMatrixMode(GL_MODELVIEW);
```

(2) 绘制位图，把位图缩放到当前窗口大小，使长宽比保持为 1:1。

```
xsize = rect->right;
ysize = BitmapInfo->bmiHeader.biHeight * xsize /
        BitmapInfo->bmiHeader.biWidth;
if (ysize > rect->bottom)
{
  ysize = rect->bottom;
  xsize = BitmapInfo->bmiHeader.biWidth * ysize /
          BitmapInfo->bmiHeader.biHeight;
};
```

值得注意的是，Windows 的 StretchBlt 函数显示位图图像的速度要比 glDrawpixels 快一些。最终的程序如下。

```
void
RepaintWindow(RECT        *rect)  /* I - Client area rectangle */
{
  GLint       xoffset,       /* X offset of image */
              yoffset;       /* Y offset of image */
  GLint       xsize,              /* X size of scaled image */
              ysize;              /* Y size of scaled image */
  GLfloat xscale,             /* Scaling in X direction */
              yscale;             /* Scaling in Y direction */

 /*
  * Reset the viewport and clear the window to white...
  */

  glViewport(0, 0, rect->right, rect->bottom);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0.0, rect->right - 1.0, 0.0, rect->bottom - 1.0, -1.0, 1.0);
  glMatrixMode(GL_MODELVIEW);

  glClearColor(1.0, 1.0, 1.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT);
 /*
  * If we have loaded a bitmap image, scale it to fit the window...
  */
```

```
 if (BitmapBits != NULL)
 {
   xsize = rect->right;
   ysize = BitmapInfo->bmiHeader.biHeight * xsize /
           BitmapInfo->bmiHeader.biWidth;
   if (ysize > rect->bottom)
   {
     ysize = rect->bottom;
     xsize = BitmapInfo->bmiHeader.biWidth * ysize /
             BitmapInfo->bmiHeader.biHeight;
   };

   xscale  = (float)xsize / (float)BitmapInfo->bmiHeader.biWidth;
   yscale  = (float)ysize / (float)BitmapInfo->bmiHeader.biHeight;

   xoffset = (rect->right - xsize) * 0.5;
   yoffset = (rect->bottom - ysize) * 0.5;

   glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
   glPixelZoom(xscale, yscale);
   glRasterPos2i(xoffset, yoffset);
   glDrawPixels(BitmapInfo->bmiHeader.biWidth,
                BitmapInfo->bmiHeader.biHeight,
          GL_RGB, GL_UNSIGNED_BYTE, BitmapBits);
 };
 glFinish();
}
```

9.4 相关函数

void glReadPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);

void glDrawPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type, GLvoid *pixels);

void glCopyPixels(GLint x, GLint y, GLsizei width, GLsizei height, GLenum format, GLenum type);

void glPixelStore{if}(GLenum pname, TYPE param);

void glPixelTransfer{if}(GLenum pname, TYPE param);

void glPixelMap{f usui}{v}(GLenum map, GLint mapsize, const TYPE *value);

void glPixelZoom(GLfloat xfactor, GLfloat yfloat);

```
void glNewList(GLuint list, GLenum mode);
void glEndList(void);
void glDeleIteLists(GLuint list, GLsizei range);
void glCallList(GLuint list);
void glCallLists(GLsizei n, GLenum type const GLvoid *lists);
GLuint glGenLists(GLsizei range);
GLboolean glIsList(GLuint list);
void glListBase(GLuint base);
```

第十章　纹理映射

使用纹理映射是过去十年来计算机图形学最显著的进步之一。OpenGL 提供了在场景中把图像与多边形相匹配的纹理映射函数。

纹理映射用于游戏中，如 DOOM，增加了房间和各种怪物的真实感。与 OpenGL 不同的是，这些游戏中用一种称为“射线造型”的方法（raycasting）把纹理图像映射到多边形上。尽管这种方法在标准图形卡上比 OpenGL 提供的纹理映射方法要快，但是，它仅局限在二维平面上，而 OpenGL 的纹理映射不受这个条件限制。

对于支持 OpenGL 和纹理映射的三维图形卡，可以不必由 CPU 完成所有纹理映射计算和准备工作，图形卡会完成这一工作，因此，速度会大幅度提高。目前市场上的这类显示卡不再是以往天文数字的价格，而是可以为多数普通用户接受了。

本章的例子可以在任何一个配有与 Windows 兼容的显示卡机器上运行。

10.1　纹理映射基础

在 OpenGL 中使用纹理映射很方便。每个纹理都是某种图像。一维纹理是有宽度、没有高度的图像，反之亦然，一维纹理的宽或高为 1 个像素。一维纹理可以替代通常的明暗处理技术并提高绘图速度。图 10-1 所示为用一维纹理显示的彩虹，纹理为彩虹上线，而获得相同效果要用七倍于此的多边形，并花费更多的绘图时间！二维纹理的高、宽均大于 1 个像素，通常由 Windows BMP 文件获得。二维纹理通常用于替代物体上复杂的几何尺寸，增加物件的真实感，如图 10-2 所示。

上面介绍的一维、二维纹理都由 RGB 颜色值构成，也可以使用色彩指数或灰度值以及 α 值。（有关内容请参见本书第十二章）

一些硬件支持 OpenGL 3D 纹理，但是一个 256×256×256 的灰度纹理要占用 16M 内存。目前，OpenGL1.1 版中包括了 3D 纹理特性作为 OpenGL 的扩展。

1．定义纹理图像

在 OpenGL 绘制纹理化多边形之前，应该定义一个纹理图像。纹理图像的存贮规则与位图相同。

2．定义一维纹理

OpenGL 的 glTexImage1D 函数用于定义一个一维纹理，

```
void glTexImage1D(GLenum target, GLint level, GLint components, GLsizei width, GLint border,
GLenum format, GLenum type, const GLvoid *pixels);
```

target 参量指定要定义的纹理种类，为 GL_TEXTURE_1D；level 标识纹理图像的层次，通常为 0，其他值用于多重纹理（见本章后面的内容）；components 参量指定用于每个像素的颜色值数目，色彩指数值为 1，RGB，RGBA 值分别为 3，4。

width 和 border 指定图像的尺寸。border 值为 OpenGL 使用的边框像素数目，可以为 0，1，2；

width 指定主纹理图像的宽度(不包括动框值) 且应该为 2 的指数。

format 参量标识颜色值的类型，可以为 GL_COLOR_INDEX，GL_LUMINANCE，GL_RGB，GL_RGBA。

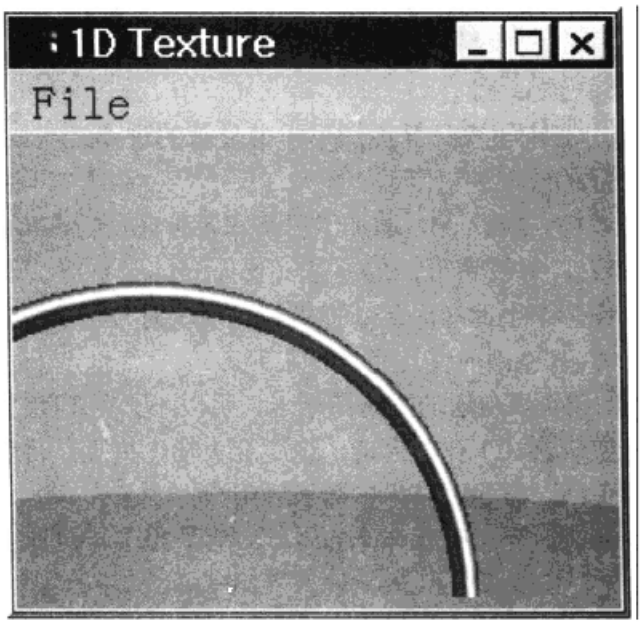


图 10-1 一维纹理

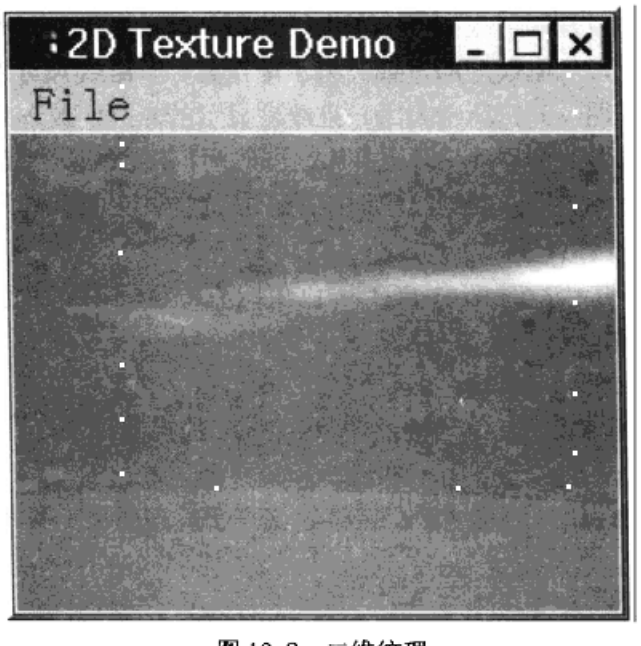


图 10-2 二维纹理

下面的程序段（程序 10-1）给出一个定义一维纹理的例子。

程序 10-1

```
void LoadAllTextures(void)
{
  static unsigned char roygbiv_image[8][3] =
  {
```

```
    { 0x3f, 0x00, 0x3f },     /* Dark Violet (for 8 colors...) */
    { 0x7f, 0x00, 0x7f },     /* Violet */
    { 0xbf, 0x00, 0xbf },     /* Indigo */
    { 0x00, 0x00, 0xff },     /* Blue */
    { 0x00, 0xff, 0x00 },     /* Green */
    { 0xff, 0xff, 0x00 },     /* Yellow */
    { 0xff, 0x7f, 0x00 },     /* Orange */
    { 0xff, 0x00, 0x00 } /* Red */
  };

  glNewList(RainbowTexture = glGenLists(1), GL_COMPILE);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage1D(GL_TEXTURE_1D, 0, 3, 8, 0, GL_RGB, GL_UNSIGNED_BYTE, roygbiv_image);
  glEndList();
}
```

创建一个包含纹理图像的显示列表。放大、缩小纹理的过滤器均为 GL_LINEAR。当纹理图像大于多边形时，用缩小过滤器；而小于多边形时，用放大过滤器。选用 GL_LINEAR 过滤器在屏幕绘图之前，OpenGL 对纹理图像的颜色值做线性插值。另外几种用于放大、缩小纹理的过滤器请参阅《精通 OpenGL》一书。

GL_NEAREST 过滤器取纹理图像中最近的像素，而不做插值运算。

3．定义二维纹理

OpenGL 用 glTexImage2D 定义二维纹理，与 glTexImage1D 函数不同的是，glTexImage2D 函数多了一个 height 参量。同样，width，height 应该是 2 的指数。

```
void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid *pixels);
```

下面的程序段（程序 10-2）读入用于绘制有云天空的纹理图像。

程序 10-2

```
void
LoadAllTextures(void)
{
  BITMAPINFO   *info;                /* Bitmap information */
  void         *bits;                /* Bitmap pixel bits */
  GLubyte *rgb;                   /* Bitmap RGB pixels */

  /*
   * Try loading the bitmap and converting it to RGB...
   */

  bits = LoadDIBitmap("sky.bmp", &info);
```

```
if (bits == NULL)
  return;

rgb = ConvertRGB(info, bits);
if (rgb == NULL)
{
  free(info);
  free(bits);

  return;
};
glNewList(SkyTexture = glGenLists(1), GL_COMPILE);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

 /*
  * Define the 2D texture image.
  */

  glPixelStorei(GL_UNPACK_ALIGNMENT, 4);  /* Force 4-byte alignment */
  glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
  glPixelStorei(GL_UNPACK_SKIP_ROWS, 0);
  glPixelStorei(GL_UNPACK_SKIP_PIXELS, 0);

  glTexImage2D(GL_TEXTURE_2D, 0, 3, info->bmiHeader.biWidth, info->bmiHeader.biHeight, 0,
               GL_RGB, GL_UNSIGNED_BYTE, rgb);
 glEndList();

/*
 * Free the bitmap and RGB images, then return 0 (no errors).
 */

 free(rgb);
 free(info);
 free(bits);
}
```

上面的两个例子均使用了显示列表存贮纹理图像。这样，可以加快绘图速度。另外，OpenGL1.1 API 的纹理化物体支持在图形硬件的纹理内存中优化存贮在显示列表中的纹理图像。

10.2 绘制纹理化的多边形

在定义纹理图像后，还要调用 glEnable 函数，允许使用纹理映射功能；否则，多边形上不出现任何纹理。如果允许进行一维纹理映射，则

```
glDisable(GL_TEXTURE_2D);
glEnable(GL_TEXTURE_1D);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

glTexEnvi 函数把纹理化设置为“印花模式”，即把纹理图像直接覆盖到多边形上。除了 GL_DECAL，还有 GL_MODULATE，GL_BLEND 模式。

GL_MODULATE 把当前纹理颜色(或亮度)与屏幕像素的颜色相乘，对于一个组合(亮度)纹理，成为一个基于纹理图像改变屏幕图像亮度的过滤器；对于 3 组分(RGB)纹理，可以获得“有色透镜”效果。与 GL_MODULATE 不同的是，GL_BLEND 把一种颜色混合到基于纹理图像的屏幕上。如果加入的颜色偏白，则看上去纹理图像好像是一团云。

在完成上面的各步骤后，就可以绘制纹理化的多边形了。下面的程序段演示如何绘制图 10-1 所示的图形。

```
glEnable(GL_TEXTURE_1D);
glCallList(RainbowTexture);
glBegin(GL_QUAD_STRIP);
  for (th = 0.0; th <= M_PI; th += (0.03125 * M_PI))
  {
    x = cos(th) * 50.0;
 y = sin(th) * 50.0;
 z = -50.0;
 glTexCoord1f(0.0);
 glVertex3f(x, y, z);

 x = cos(th) * 55.0;
 y = sin(th) * 55.0;
 z = -50.0;
 glTexCoord1f(1.0);
 glVertex3f(x, y, z);
  };
glEnd();
```

函数 glTexCoord 布置纹理图像的位置。0.0 代表图像最左边的像素，1.0 为最右边像素。如果超出这一范围的值，就要根据 GL_TEXTURE_WRAP_S 参数的值做不同的处理。在缺省情况下为 GL_CLAMP，纹理坐标要限制在 0.0～1.0 范围内（如图 10-3 所示）；若用 GL_REPEAT 参数，则在一定间隔后会重复纹理图像（如图 10-4 所示）。纹理坐标用 S,T(或 s,t)表示，而不用 X,Y 表示。

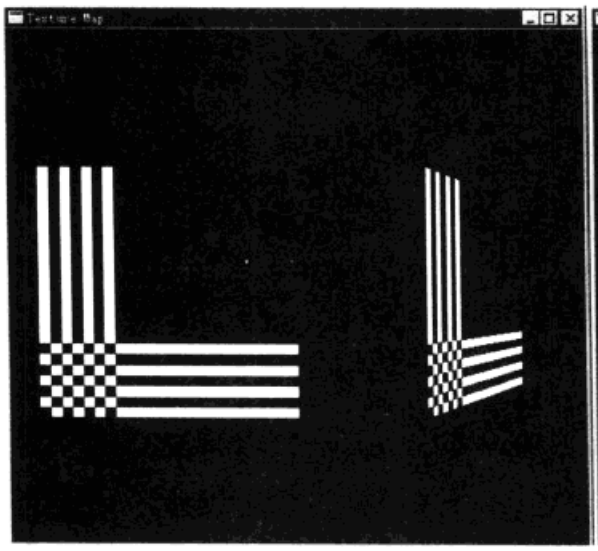


图 10-3　GL_CLAMP

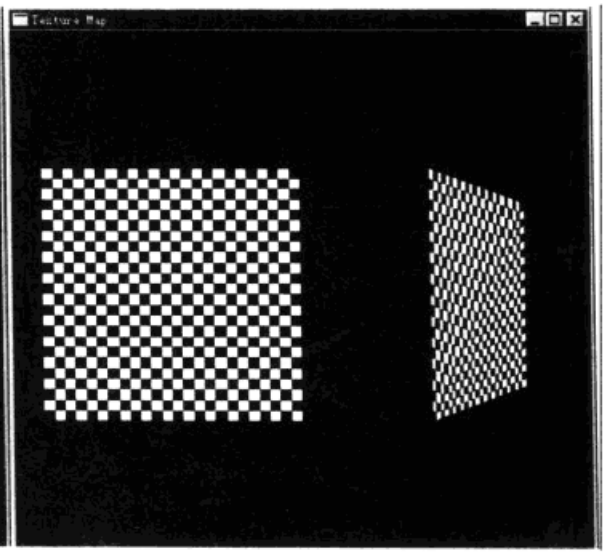


图 10-4　GL_REPEAT

10.3　多重纹理

在上面所举例子中所用的纹理仅有一个层次，但在做动画时，需要跟据与观察点的距离，给出不同层次的纹理，例如，在欣赏一幅油画时，需要很高的分辨率，而距离很远时，只要绘出一个轮廓就可以了。

OpenGL 支持称为多重纹理的多层次纹理，可以不必使用 GL_LINEAR 图像过滤器改进绘图性能。要为多重纹理指定各层次的图像，把上面的例子中纹理定义改为：

```
static unsigned char roygbiv_image0[16][3];
static unsigned char roygbiv_image1[8][3];
static unsigned char roygbiv_image2[4][3];
static unsigned char roygbiv_image3[2][3];
 static unsigned char roygbiv_image4[1][3];
 glNewList(RainbowTexture=glGenLists(1), GL_COMPILE);
      glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
      glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                                              GL_NEAREST_MIPMAP_LINEAR);

      glTextureImage1D(GL_TEXTURE_1D, 0, 3, 16, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                          roygbiv_image0);
      glTextureImage1D(GL_TEXTURE_1D, 0, 3, 8, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                          roygbiv_image1);
      glTextureImage1D(GL_TEXTURE_1D, 0, 3, 4, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                          roygbiv_image2);
      glTextureImage1D(GL_TEXTURE_1D, 0, 3, 2, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                          roygbiv_image3);
```

```
    glTextureImage1D(GL_TEXTURE_1D, 0, 3, 1, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                        roygbiv_image4);
glEndList();
```

层次 0 为纹理的最高分辨率图像，层次 1 的分辨率为层次 0 尺寸的一半，依次类推。在用多重纹理绘制多边形时，要用 GL_TEXTURE_MIN_FILTER 之一。GL_TEXTURE_MIN_FILTER 包括 GL_NEAREST_MIPMAP_NEAREST，GL_LINEAR_MIPMAP_NEAREST，GL_LINEAR_MIPMAP_LINEAR，GL_NEAREST_MIPMAP_LINE。

使用 GL_LINER_MIPMAP_NEAREST 和 GL_LINEAR_MIPMAP_LINEAR 过滤器时，会牺牲一些显示性能。GL_NEAREST_MIPMAP_NEREST 过滤器在性能上与 GL_NEAREST 相当，但通常会获得很好的效果。选择多重图像要比较所绘制的多边形与多重图像的尺寸大小。为此，OpenGL 的应用库(GLU32.LIB)提供两个函数 gluBuild1DMipmaps 和 gluBuild2DMipmaps，根据最高分辨率图像生成多重图像。上面的程序可以修改为：

```
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                                        GL_NEAREST_MIPMAP_LINEAR);
gluBuild1DMipmaps(GL_TEXTURE_1D, 3, 8, 0, GL_RGB, GL_UNSIGNED_BYTE,
                                                    roygbiv_image);

/* 2D texture */
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                                        GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, info->bmiHeader.biWidth,
            info->bmiHeader.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, rgb);
```

注意：因为上面提到的两个函数根据一个图像生成一系列图像，有的纹理图像可能不够准确，这时，就要用户自己创建这个图像。

10.4 自动生成纹理坐标

除了用 glTexCoord 命令为纹理坐标赋值外，OpenGL 还可以用 glTexGen 命令自动生成纹理坐标，这个功能在显示轮廓线、环境映射等许多方面都很有用处。

```
void glTexGen{ifd}{v} (GLenum coord, GLenum pname, TYPE param);
```

coord 为要生成的坐标 GL_S，GL_T，GL_R，GL_Q，分别代表纹理坐标的 s，t，r，q；pname 的值为 GL_TEXTURE_GEN_MODE，GL_OBJECT_PLANE，GL_EYE_PLANE，对于 GL_TEXTURE_GEN_MODE，param 是 GL_OBJECT_LINEAR，GL_EYE_LINEAR 或 GL_SPHERE_MAP，是生成纹理坐标的函数。

1. 创建轮廓线

若选用 GL_TEXTURE_GEN_MODE 和 GL_OBJCT_LINEAR，生成函数是物体顶点坐标(x_0，y_0，z_0，w_0)的线性组合。

生成坐标=$p_1x_0+p_2y_0+p_3z_0+p_4w_0$

p_1, p_2, p_3, p_4 为 pname 是 GL_OBJECT_PLANE 时 param 的值，若 p_1, p_2, p_3, p_4 是归一化的，得到的结果为顶点到平面的距离，在平面的一侧为正，另一侧为负，顶点在平面上，则距离为零。

程序 10-3 在一个方块上绘制带状纹理，应该注意的是这里的纹理坐标在物体坐标系内，而不是视觉坐标，参见图 10-5，这里仅生成了 s 坐标，在二维纹理映射时，也可以分别生成 s, t 坐标。

程序 10-3

```
#include "windows.h"
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glaux.h>

void myinit(void);
void makeStripeImage(void);
void CALLBACK display(void);
void CALLBACK myReshape(GLsizei w, GLsizei h);

#define stripeImageWidth 64
GLubyte stripeImage[3*stripeImageWidth];
void makeStripeImage(void)
{
    int j;

    for (j = 0; j < stripeImageWidth; j++) {
         stripeImage[3*j] =  255;
        stripeImage[3*j+1] =255-2*j;
        stripeImage[3*j+2] =255;

      }
}

/* glTexGen stuff: */

GLfloat sgenparams[] = {1.0, 1.0, 1.0, 0.0};

void myinit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);

    makeStripeImage();
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

```
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage1D(GL_TEXTURE_1D, 0, 3, stripeImageWidth, 0,
         GL_RGB, GL_UNSIGNED_BYTE, stripeImage);

    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, sgenparams);

    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_1D);
    glEnable(GL_CULL_FACE);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glFrontFace(GL_CW);
    glCullFace(GL_BACK);
    glMaterialf (GL_FRONT, GL_SHININESS, 64.0);
}

void CALLBACK display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glRotatef(25.0, 1.0, 0.0, 0.0);
    auxSolidTeapot(1.5);
    glPopMatrix ();
    glFlush();
}

void CALLBACK myReshape(GLsizei w, GLsizei h)
{
    float a=3.5;

    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
    if (w <= h)
    glOrtho (-a, a, -a*(GLfloat)h/(GLfloat)w,
        a*(GLfloat)h/(GLfloat)w, -a, a);
    else
    glOrtho (-a*(GLfloat)w/(GLfloat)h,
        a*(GLfloat)w/(GLfloat)h, -a, a, -a, a);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void main(void)
{
    auxInitDisplayMode (AUX_SINGLE | AUX_RGB | AUX_DEPTH16);
    auxInitPosition (0, 0, 500, 500);
    auxInitWindow (" Cube TextureMapping");
    myinit();
    auxReshapeFunc (myReshape);
    auxMainLoop(display);
 }
```

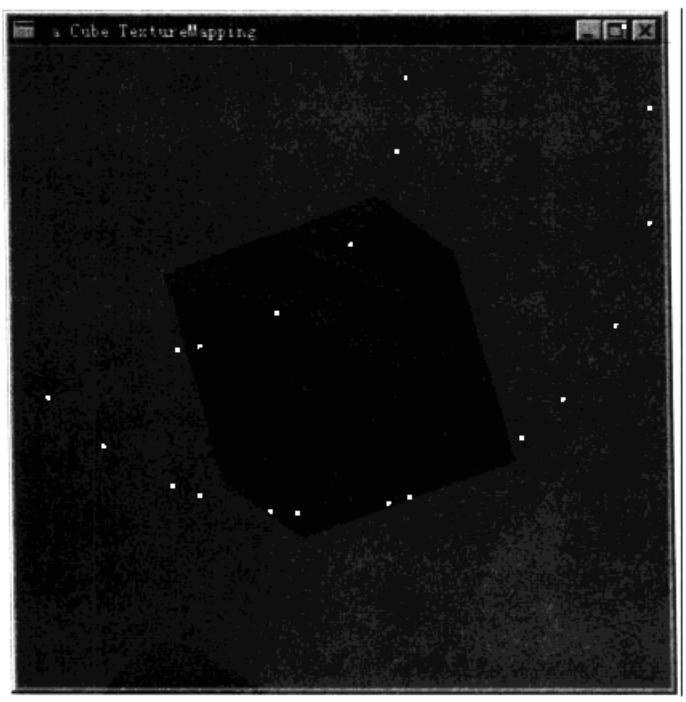


图 10-5　glGenTex 函数的使用

2. 环境映射

进行环境映射的目的是绘制一个完全反射的物体，使观察者在其表面看到周围的环境，而看到的场景取决于观察点与物体的相对位置和物体表面形状。为进行环境映射，需要创建一个适当的纹理映射，并利用 OpenGL 自动生成纹理坐标。

环境映射的基本假设是环境中的物体要对于反光体表面足够远，反射光的方向决定绘制的颜色，确定平直的纹理映射各方向的颜色等价于在环境中放入一个抛光的球体，并用放在远处的长焦距镜头照相机对其拍照，得到一个圆形的纹理映射区域。为得到这一效果，可以扫描一张特别宽角度镜头拍摄的照片，然后，调用 OpenGL 环境映射算法，以找到球面上与物体有相同切面的点，并在相应点处绘制球面上可见的颜色。编程时，只需把

```
glTexGenfv (GL_S, GL_OBJECT_LINEAR, sgenparams);
glEnable (GL_TEXTURE_GEN_S);
```

改为

```
glTexGenfv (GL_S, GL_SPHERE_MAP , 0);
glTexGenfv (GL_T, GL_SPHERE_MAP, 0);
glEnable (GL_TEXTURE_GEN_S);
glEnable (GL_TEXTURE_GEN_T);
```

用 GL_SPHERE_MAP 分别为 s，t 方向调用环境映射。

10.5 相关函数

```
void glTexParameter{if}{v}(GLenum target, GLenum pname, TYPE param);
void glTexEnv{if}{v}(GLenum target, GLenum pname, TYPE param);
void glTextCoord1{sifd}{v}(TYPE s);
void glTextCoord2{sifd}{v}(TYPE s, TYPE t);
void glTextCoord3{sifd}{v}(TYPE s, TYPE t, TYPE r);
void glTextCoord4{sifd}{v}(TYPE s, TYPE t, TYPE r, TYPE q);
void glTexGen{ifd}{v}(GLenum coord, GLenum pname, TYPE param);
void glTexImage1D(GLenum target, GLint level, GLint components, GLsizei width,  GLint border,
GLenum format, GLenum type, const GLvoid *pixels);
void glTexImage2D(GLenum target, GLint level, GLint components, GLsizei width, GLsizei height,
GLint border, GLenum format, GLenum type, const GLvoid *pixels);
```

第十一章　绘制三维物体

本章讲述创建光滑曲线、曲面的方法，其内容由两部分组成：

①创建二次曲面；

②绘制NURBS曲线、曲面。

对于多数三维图形程序来说，需要创建出光滑的曲面和曲线，而前面介绍的曲线、曲面都是用近似方法得到的。例如，把一个曲面划分成许多小三角形、四边形，从而得到一个近似曲面；把一个曲线用很多线段连接而成。采用上述方法不仅浪费时间，也带来一些误差。

Pierre Bezier在70年代，创建了一系列用少量控制点表示曲线、曲面的数学模型，从而大大简化了创建曲线、曲面的方法，并且提高了精度。本章将讲述创建Bezier曲线、曲面以及NURBS曲线、曲面的方法。

本章的另一个内容是介绍如何使用OpenGL应用库函数绘制二次曲面。绘制二次曲面的函数是OpenGL应用库(GLU32.LIB)的一部分，可以绘制圆锥、圆柱、盘、圆球等几何形状。本章将讲述如何在程序中使用这些函数。

11.1　创建一个二次曲面

在窗口上绘制的二次曲面都有一个相关联的绘图状态，gluNewQuadric函数创建一个描述当前绘图风格、方位、光照模式、纹理模式、回调函数的不透明状态变量。

```
GLUquadricObj *obj
obj = gluNewQuadric();
```

值得注意的是，二次曲面的状态不包括要绘制的几何形状，而是描述如何绘制这些几何形状。可以在绘制许多不同类型形状时重复使用二次曲面。

1．改变绘制二次曲面的方式

在创建一个二次曲面后，可以调用gluQuadricDrawStyle，gluQuadricNormals，gluQuadricOrientation，gluQuadricTexture函数改变二次曲面的状态，由用户设定绘图的形状。

gluQuadricDrawStyle选择绘制OpenGL几何要素的风格，缺省值为用多边形和带状排列几何要素填充该几何形状，GL_FILL。表11-1为绘制二次曲面的风格选项。

表11-1　二次曲面的风格

风　格	解　释
GLU_FILL	绘制填充的二次曲面
GLU_LINE	用一系列线段绘制二次曲面的线框图
GLU_SILHOUETTE	用一系列线段绘制二次曲面的外缘
GLU_POINT	用一系列点绘制二次曲面

OpenGL自动生成光照处理用的法线，表11-2所示为二次曲面的法线模式。

表11-2 二次曲面的法线模式

法线模式	解 释
GLU_NONE	不生成法线
GLU_FLAT	为每个多边形生成法线，由多个平面构成的表面
GLU_SMOOTH	为每个顶点生成法线，得到光滑表面

gluQuadricOrientation函数控制法线的方向，GLU_OUTSIDE使法线朝外，GLU_INSIDE指向曲面内部。

OpenGL自动为二次曲面生成纹理坐标。gluQuadricTexture函数的GL_TRUE，GL_FALSE参数分别对应是否生成纹理。

2. 绘制圆柱

gluCylinder函数用于绘制圆柱。可以把圆柱看成沿z轴伸展两端不封闭的管子，如图11-1所示。

```
void gluCylinder(GLUquadricObj *obj, GLdouble baseRadius, GLdouble topRadius,
                                                GLint slices, GLint stacks);
```

baseRadius和topRadius分别指定圆柱底、顶的半径，height为圆柱的高度。slice和stack分别定义圆柱柱面沿圆周和沿高度等分数目。要得到光滑柱面，这个等分的数目在20左右。如果使用聚光灯或镜面高光，stack值要比较大，通常与height值相同，否则，使stack为2。

如果取baseRadius为0，则可以绘制一个圆锥。对一个圆柱进行纹理化时，纹理从圆柱的前缘(0, radius, 0)开始缠绕，因此，要把纹理图像颠倒过来。

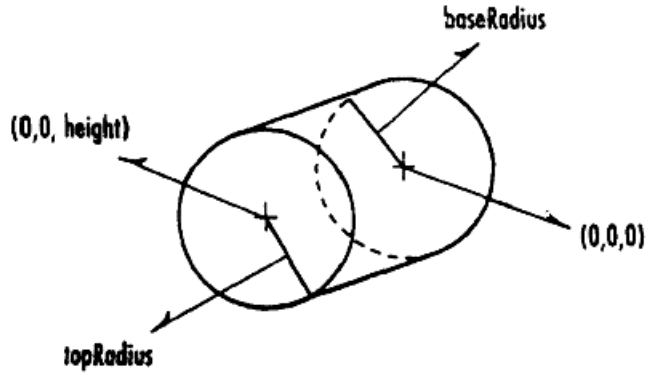


图11-1 圆柱参数示意图

3. 绘制圆盘

绘制圆盘的函数为

```
void gluDisk(GLUquadricObj *obj, GLdouble innerRadius, GLdouble outerRadius,
            GLint slices, GLint loops);
```

innerRadius、outerRadius分别为圆盘的内径、外径。若innerRadius为0，则绘制一个实心圆。slice参数定义等分圆周的数目，选用20左右的值，才能使圆盘的内、外圆形状近似为圆。loops参数为内径与外径间同心圆的数目，如果要改善聚光灯、镜面光照的效果，就要取loops较大的值。

纹理图像映射到内径与外径之间环形区域内，纹理图像的顶部映射到圆盘外径，纹理图像左侧映射到圆盘的左侧。

4. 绘制部分圆盘

OpenGL应用库中的gluPartialDisk函数绘制圆盘的一部分。

void gluPartialDisk(GLUquadricObj *obj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops, GLdouble startAugle, GLdouble sweepAngle);

startAngle指定圆盘顶部顺时针方向的角度，sweepAngle指定所绘制扇形的角度。

5. 绘制圆球

void gluSphere(GLUquadricObj *obj, GLdouble radius, GLint slices, GLint stacks);

其中slice参量为圆球经线数目，stack为纬线数目。

用经线、纬线坐标把纹理图像映射到圆球上。

11.2 Bezier 曲线和曲面

1. 参数方程

在几何学中，三维空间的曲线和曲面都可以用参数方程表示，如图11-2所示，其中描述曲线的参数方程只有一个自变量u，而描述曲面的参数方程有两个自变量u, v。

在OpenGL中定义一条曲线，也要给出该曲线的参数方程。曲线由参数u及其取值范围确定。同样，由参数方程的两个参数描述一个曲面。值得注意的是，这些描述曲线、曲面的参数只反映其在空间变化趋势而不反映实际的坐标值。

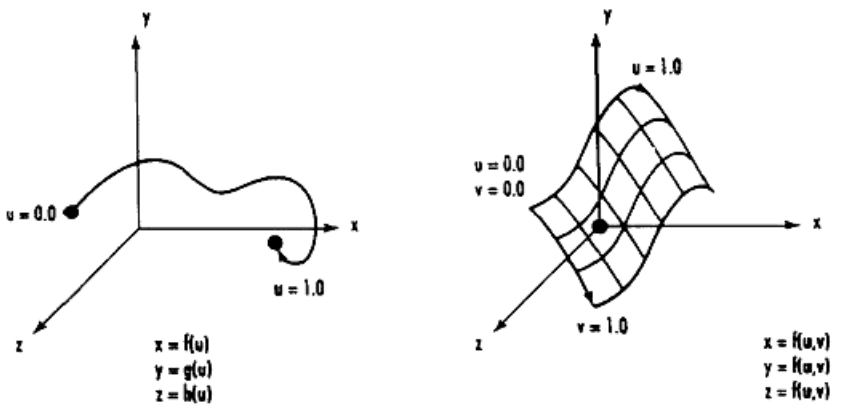


图11-2 曲线、曲面的参数方程

2. 控制点

控制点描述曲线的形状。对于Bezier曲线，其开始控制点和终止控制点在曲线上。另外一些控制点的作用相当于磁铁，把曲线拉过去，如图11-3所示。用于描述曲线形状的控制点数目也就是曲线的阶数，而曲线的自由度是阶数减去一。这些名词的数学含义与描述曲线的参数方程有关。阶数为其系数的数目，而自由度为参数的最高阶指数。由此得出结论，图11-3中的(b)为二次曲线，(c)为三次曲线。理论上可以定义任意阶曲线，但阶数过高，曲线形状会因控制点轻微移动而发生剧烈振荡，所以，在OpenGL中最常用的Bezier曲线为3次曲线。

3. 连续性

曲线的连续性描述两条曲线相接处的光滑过渡程度，如图11-4所示可以把分段曲线连续性分为

(1) C^0——两曲线不相交

(2) C^1——两曲线有一个公共端点

(3) C^2——两曲线有一个公共切线

(4) C^3——两曲线有一个公共曲率

用分段构造复杂曲线、曲面时，通常要具有C^2或C^3阶连续性。

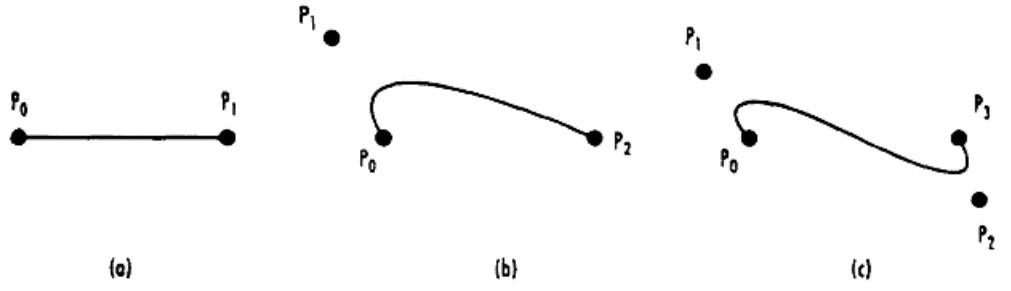


图11-3　控制点对曲线形状的影响

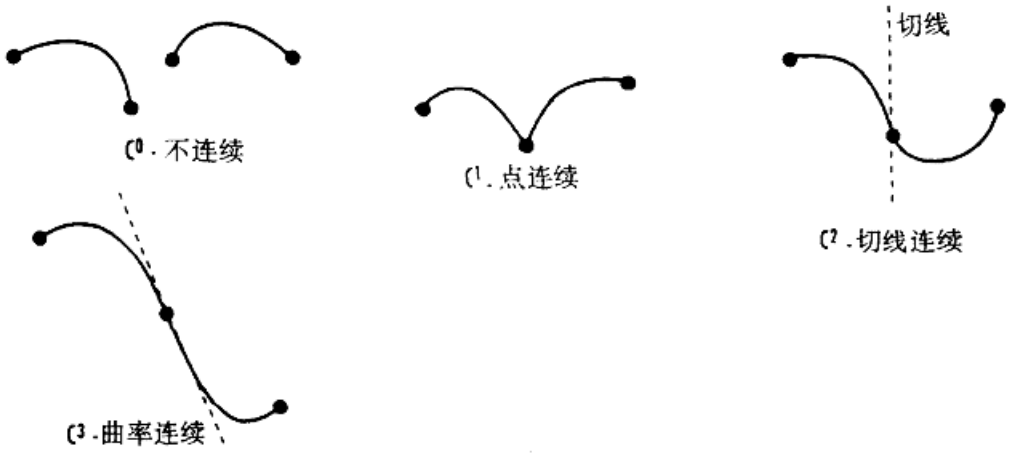


图11-4　分段曲线的连续性

4. 求值器

OpenGL提供了通过指定控制点、描述曲线、曲面参数的取值范围来绘制Bezier曲线、曲面的函数。调用这些求值函数，可以生成构造曲线、曲面的点。下面介绍一个绘制二维Bezier曲线的例子，然后把这个例子扩展到三维空间，创建一个Bezier曲线。

①二维曲线

程序11-1为创建二维Bezier曲线程序的一部分，用于定义Bezier曲线的控制点并用求值器绘制曲线。输出结果如图11-5所示。

程序11-1

```
// The number of control points for this curve
GLint nNumPoints = 4;

GLfloat ctrlPoints[4][3]= {{  -4.0f, 0.0f, 0.0f}, // End Point
                           { -6.0f, 4.0f, 0.0f},      // Control Point
                            { 6.0f, -4.0f, 0.0f},     // Control Point
                            { 4.0f, 0.0f, 0.0f }};    // End Point
```

```
// Declaration for Window procedure
LRESULT CALLBACK WndProc(    HWND     hWnd,
                                                    UINT     message,
                                                    WPARAM   wParam,
                                                    LPARAM   lParam);

// Dialog procedure for about box
BOOL APIENTRY AboutDlgProc (HWND hDlg, UINT message, UINT wParam, LONG lParam);

// Set Pixel Format function - forward declaration
void SetDCPixelFormat(HDC hDC);
void RenderScene(void);

// This function is used to superimpose the control points over the curve
void DrawPoints(void)
        {
        int i;  // Counting variable

        // Set point size larger to make more visible
        glPointSize(5.0f);

        // Loop through all control points for this example
        glBegin(GL_POINTS);
                for(i = 0; i < nNumPoints; i++)
                        glVertex2fv(ctrlPoints[i]);
        glEnd();
        }

// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
        {
        // Prevent a divide by zero
        if(h == 0)
                h = 1;

        // Set Viewport to window dimensions
    glViewport(0, 0, w, h);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        gluOrtho2D(-10.0f, 10.0f, -10.0f, 10.0f);
```

```
        // Modelview matrix reset
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        }

// This function does any needed initialization on the rendering
// context.  Here it sets up and initializes the lighting for
// the scene.
void SetupRC()
        {
        // Clear Window to white
        glClearColor(1.0f, 1.0f, 1.0f, 1.0f );

        // Draw in Blue
        glColor3f(0.0f, 0.0f, 1.0f);
        }

// Called to draw scene
void RenderScene(void)
        {
        //int i;

        // Clear the window with current clearing color
        glClear(GL_COLOR_BUFFER_BIT);

        // Sets up the bezier
        // This actually only needs to be called once and could go in
        // the setup function
        glMap1f(GL_MAP1_VERTEX_3,         // Type of data generated
        0.0f,                             // Lower u range
        100.0f,                  // Upper u range
        3,                                // Distance between points in the data
        nNumPoints,                                    // number of control points
        &ctrlPoints[0][0]);                      // array of control points

        // Enable the evaluator
        glEnable(GL_MAP1_VERTEX_3);

        // Use a line strip to "connect-the-dots"
        glBegin(GL_LINE_STRIP);
```

```
        for(i = 0; i <= 100; i++)
                {
                // Evaluate the curve at this point
                glEvalCoord1f((GLfloat) i);
                }
    glEnd();

    // Use higher level functions to map to a grid, then evaluate the
    // entire thing.

    // Map a grid of 100 points from 0 to 100
    glMapGrid1d(100,0.0,100.0);

    // Evaluate the grid, using lines
    glEvalMesh1(GL_LINE,0,100);

    // Draw the Control Points
    DrawPoints();

    // Flush drawing commands
    glFlush();
    }
```

程序段开始时，定义曲线的控制点：

```
// The number of control points for this curve
GLint nNumPoints = 4;

GLfloat ctrlPoints[4][3]= {{  -4.0f, 0.0f, 0.0f}, // End Point
                           { -6.0f, 4.0f, 0.0f},     // Control Point
                            { 6.0f, -4.0f, 0.0f},    // Control Point
                            { 4.0f, 0.0f, 0.0f }};   // End Point
```

DrawPoints函数用于在屏幕上显示这些控制点，这对理解控制点的作用是很有益处的。ChangSize函数创建一个X、Y方向取值范围均为[-10，+10]的二维正交投影。

在RenderScene函数中，首先调用glMap1f函数创建曲线的映射。

```
// Called to draw scene
void RenderScene(void)
    {
    //int i;

    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);
```

```
// Sets up the bezier
// This actually only needs to be called once and could go in
// the setup function
glMap1f(GL_MAP1_VERTEX_3,          // Type of data generated
0.0f,                              // Lower u range
100.0f,                  // Upper u range
3,                                 // Distance between points in the data
nNumPoints,                                        // number of control points
&ctrlPoints[0][0]);                          // array of control points

// Enable the evaluator
glEnable(GL_MAP1_VERTEX_3);
```

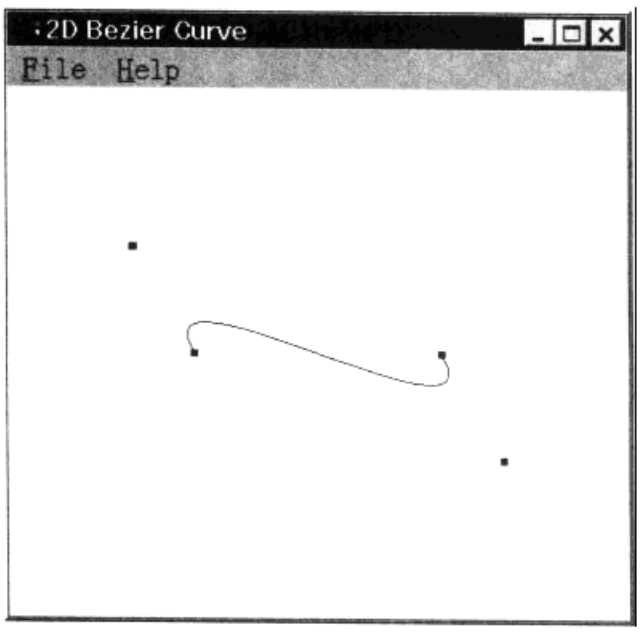


图 11-5 Bezier 曲线及控制点

glMap1f的第一个参数GL_MAP1_VERTEX_3用求值器生成顶点坐标(x,y,z)， 也可以用求值器生成其他值，如纹理坐标、颜色等信息。其后两个参数分别指定参数u 的下限、上限，对应曲线的开始点和终止点。3为控制点数组的顶点浮点值数目。每个顶点由(x,y,z)三个浮点值构成。nNumPoints参数给出沿曲线的求值点数目。

最后一个参数为曲线控制点缓冲区的指针。在创建了这一映射后，可以在求值器中使用该映射。下面的调用允许用求值沿曲线上生成点。

```
// Enable the evaluator
glEnable(GL_MAP1_VERTEX_3);
```

glEvalCoord1f的唯一参数是曲线的系数，用这一系数对曲线求值并在内部为该点调用glVertex函数。对曲线的取值范围内的每点调用glEvalCoord函数，生成曲线上的顶点坐标，可以用这些顶点的连线绘制出该曲线。

```
// Use a line strip to "connect-the-dots"
glBegin(GL_LINE_STRIP);
```

```
        for(i = 0; i <= 100; i++)
                {
                // Evaluate the curve at this point
                glEvalCoord1f((GLfloat) i);
                }
    glEnd();
```

最后，在窗口中显示这些控制点：

```
    // Draw the Control Points
    DrawPoints();

    // Flush drawing commands
    glFlush();
```

②对曲线求值

OpenGL的glMapGrid函数用于创建在取值范围内沿曲线均匀布置的点，然后，调用glEvalMesh函数把这些点连接起来。下面的两段程序是等价的。

程序段1

```
//Use higher level functions to map to a grid, then evaluate the entire thing

//Map a grid of 100 points from 0 to 100
glMapGrid(GL_LINE, 0, 100);

//Evaluate the grid, using lines
glEvalMesh(GL_LINE, 0, 100);
```

程序段2

```
    // Use a line strip to "connect-the-dots"
    glBegin(GL_LINE_STRIP);
        for(i = 0; i <= 100; i++)
                {
                // Evaluate the curve at this point
                glEvalCoord1f((GLfloat) i);
                }
    glEnd();
```

③三维曲面

创建三维Bezier曲线与二维曲线的情形相似，只是除了要定义沿u方向的点，还要定义沿v方向的点。程序11-2显示一个三维Bezier曲面的线框图。开始时，要定义3组沿v方向的控制点。为简便起见，不同z值上相应的控制点均相同。这样，得到一个均匀曲面，相当于在z轴上拉伸一个二维Bezier曲线。

程序 11-2

```
// The number of control points for this curve
```

```
GLint nNumPoints = 3;

GLfloat ctrlPoints[3][3][3]= {{{  -4.0f,  0.0f,  4.0f},
                                { -2.0f,  4.0f,  4.0f},
                                {  4.0f,  0.0f,  4.0f }},

                               {{  -4.0f,  0.0f,  0.0f},
                                { -2.0f,  4.0f,  0.0f},
                                {  4.0f,  0.0f,  0.0f }},

                               {{  -4.0f,  0.0f, -4.0f},
                                { -2.0f,  4.0f, -4.0f},
                                {  4.0f,  0.0f, -4.0f }}};
......
......
// Called to draw scene
void RenderScene(void)
        {
        //int i;

        // Clear the window with current clearing color
        glClear(GL_COLOR_BUFFER_BIT);

        // Save the modelview matrix stack
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();

        // Rotate the mesh around to make it easier to see
        glRotatef(45.0f, 0.0f, 1.0f, 0.0f);
        glRotatef(60.0f, 1.0f, 0.0f, 0.0f);

        // Sets up the Bezier
        // This actually only needs to be called once and could go in
        // the setup function
        glMap2f(GL_MAP2_VERTEX_3,        // Type of data generated
        0.0f,                            // Lower u range
        10.0f                            // Upper u range
        3,                               // Distance between points in the data
        3,                               // Dimension in u direction (order)
        0.0f,                            // Lover v range
        10.0f,                           // Upper v range
```

```
9,                              // Distance between points in the data
3,                              // Dimension in v direction (order)
&ctrlPoints[0][0][0]);          // array of control points

// Enable the evaluator
glEnable(GL_MAP2_VERTEX_3);

// Use higher level functions to map to a grid, then evaluate the
// entire thing.

// Map a grid of 10 points from 0 to 10
glMapGrid2f(10, 0.0f, 10.0f, 10, 0.0f, 10.0f);

// Evaluate the grid, using lines
glEvalMesh2(GL_LINE, 0, 10, 0, 10);

// Draw the Control Points
DrawPoints();

// Restore the modelview matrix
glPopMatrix();

// Flush drawing commands
glFlush();
}
```

在RenderScene函数中，glMap2f定义沿u,v两个方向上的控制点，而不再是上一个例子中glMap1f仅定义u方向的控制点。

```
// Sets up the Bezier
// This actually only needs to be called once and could go in
// the setup function
glMap2f(GL_MAP2_VERTEX_3,       // Type of data generated
0.0f,                           // Lower u range
10.0f                           // Upper u range
3,                              // Distance between points in the data
3,                              // Dimension in u direction (order)
0.0f,                           // Lover v range
10.0f,                          // Upper v range
9,                              // Distance between points in the data
3,                              // Dimension in v direction (order)
&ctrlPoints[0][0][0]);          // array of control points
```

值得注意的是，glMap2f函数要指定生成数据的类型，u方向取值的上限、下限，u方向

数据间隔，u方向的维数，v以及存贮控制点的数组。尽管在u方向数据间隔仍为3，但是v方向的数据间隔应该为9，即u,v两个方向生成数据数目相乘的结果(3×3=9)。

与二维求值器相似，应该先调用glEnable函数，再调用glMapGrid2f函数给出u,v两方向等分的数目。

```
// Enable the evaluator
glEnable(GL_MAP2_VERTEX_3);

// Use higher level functions to map to a grid, then evaluate the
// entire thing.

// Map a grid of 10 points from 0 to 10
glMapGrid2f(10,0.0f,10.0f,10,0.0f,10.0f);
```

设置了求值器后，调用 glEvalMesh2 对曲面网格求值。程序的显示结果如图 11-6 所示。

```
// Evaluate the grid, using lines
glEvalMesh2(GL_LINE,0,10,0,10);
```

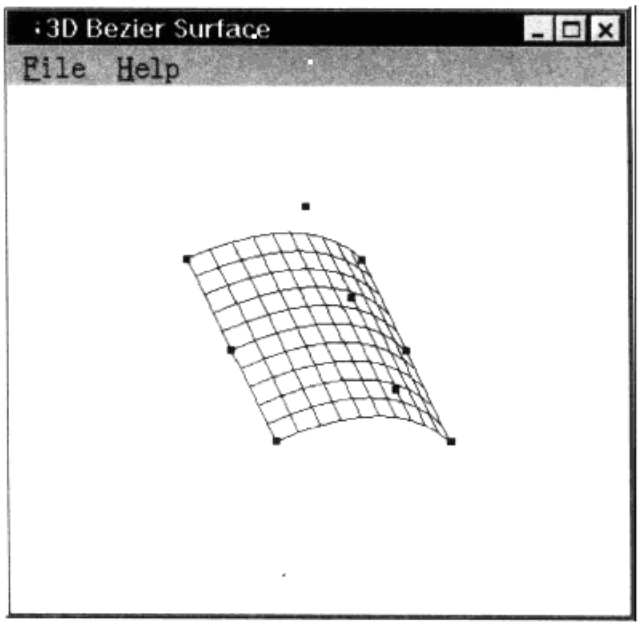


图 11-6 Bezier 曲面

④光照及法向矢量

如果利用求值器自动生成曲面的法线，仅仅需要把上面两行程序修改为：

```
// Evaluate the grid, using lines
glEvalMesh2(GL_LINE,0,10,0,10);
```

然后，调用下面的函数：

```
glEnable(GL_AUTO_NORMAL);
```

图11-7所示为自动生成法线后，光照处理的结果。

图11-7　经过光照处理的曲面

11.3　NURBS

尽管可以用求值器求解任意自由度的Bezier曲面，但是在OpenGL中绘制非常复杂的曲面时，还要通过把Bezier曲面分片组合起来的方法来实现。如果控制点过多，则曲线的连续性又无法保证。为此，OpenGL的GLU库提供了NURBS函数解决这一问题。NURBS是Non-Uniform-Rational B-Spline的缩写，即为不均匀有理B样条。NURBS是Bezier及其他许多形式的曲线、曲面更普遍的形式，可以用大量控制点生成光滑的曲线、曲面。

1．从Bezier到B样条

用两个控制点作为两个端点，再用其他控制点改变曲线的形状，就可以达到控制Bezier曲线的目的。图11-8所示为3，4，5个控制点定义的三条Bezier曲线。一点切线方向为该点与相近点的连线。2次、3次Bezier曲线相当光滑，通常为C^3连续。对于更多的控制点光滑性反而会下降。

B样条(双3次样条)把曲线分段，每一段的形状仅影响相近的4个控制点。这样组合后得到的曲线类似4阶Bezier曲线。尽管控制点比较多，但每一线段的结合处有C^3连续性。也就是说，这一曲线不必通过所有控制点。

2．结点

NURBS强大的功能表现在，对于任一曲线给定线段，可以绕过4个控制点的影响而得任意光滑的曲线。上述操作是由一系列称为结点的值得到的。

对每一控制定义两个结点值。这些值是单调递增且与u,v参数区域相匹配。因为结点值决定了u,v空间取值范围内控制点的影响。图11-9所示为u方向4个单位长度曲线上控制点的影响。u取值范围中间值对曲线的影响最大，且只有0～3之间的点才能影响曲线的形状。结点序列定义了在取值范围内影响点的强度。若重复一个结点值，则这一参数值附近点的影响加大。这种重复称为结点的重数。一个区域内，重数越高，曲线或曲面的曲率越小。

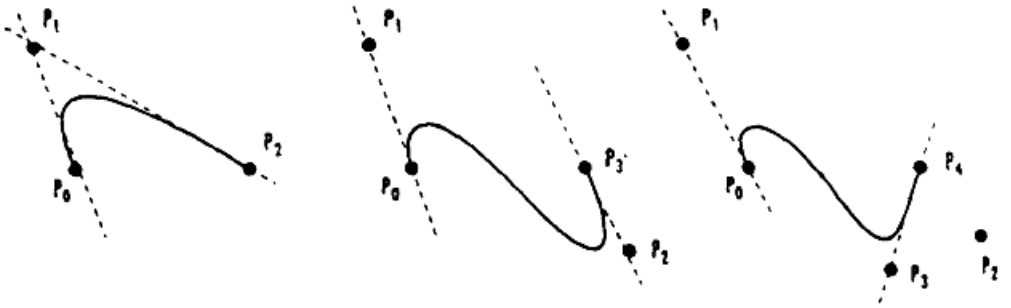


图11-8 Bezier曲线连续性与阶数的关系

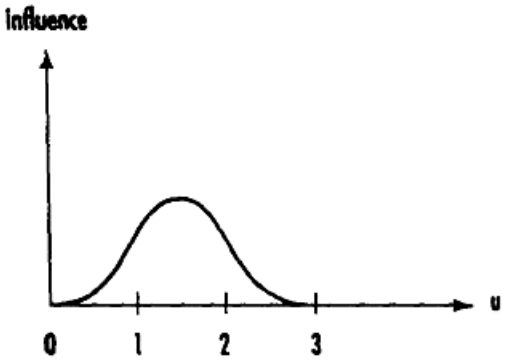


图11-9 控制点的影响

3. 创建一个NURBS曲面

GLU库提供了非常方便的绘制NURBS曲面的高级函数，使用户不必调用求值器或建立映射、网格。GLU库提供了两个函数：gluNewNurbsRenderer函数提供的NURBS绘图工具绘制NURBS对象，而gluDeleteNurbsRenderer函数删除这个NURBS对象。下面的程序段说明如何使用这些函数。

```
// NURBS object pointer
GLUnurbsObj *pNurb = NULL;
.....
// Setup the Nurbs object
pNurb = gluNewNurbsRenderer();
.....
// Do your NURBS
.....
// Delete the Nurbs objectif it was created
if (pNurbs)
        gluDeleteNurbsRenderer(pNurbs);
```

①NURBS属性

在创建了NURBS绘图工具后，还要为NURBS设置属性。

```
// Set sampling tolerance
gluNurbsProperty(pNurb, GLU_SAMPLING_TOLERANCE, 25.0f);
```

```
// Fill to make a solid surface (use GLU_OUTLINE_POLYGON to create a polygon mesh)
gluNurbsProperty(pNurb, GLU_DISPLAY_MODE, (GLfloat)GLU_FILL);
```

其中GLU_SAMPLING_TOLERANCE定义曲面上网格的密度，GLU_FILL对生成的网格做填充处理，而不仅仅实现框图。

②定义曲面

gluNurbsSurface函数的控制点和结点序列数组定义曲面的形状。这一函数包括在glBeginSurface/glEndSurface之间:

```
    // Render the NURB
    // Begin the NURB definition
gluBeginSurface(pNurb);

    // Evaluate the surface
gluNurbsSurface(pNurb,        // pointer to NURBS renderer
    8, Knots,                 // No. of knots and knot array u direction
    8, Knots,                 // No. of knots and knot array v direction
    4 * 3,                    // Distance between control points in u dir.
    3,                        // Distance between control points in v dir.
    &ctrlPoints[0][0][0], // Control points
    4, 4,                     // u and v order of surface
    GL_MAP2_VERTEX_3);        // Type of surface

    // Done with surface
    gluEndSurface(pNurb);
```

可以用 gluNurbsSurface 创建任意多个 NURBS 曲面，但是用 gluNurbsProperty 函数定义的 NURBS 曲面属性在程序中一直在起作用。

下面是用控制点和结点值生成 NURBS 曲面程序的一部分。输出的结果如图 11-10 所示。

```
// Mesh // Mesh extends four units -6 to +6 along x and y axis
// Lies in Z plane
//                u  v  (x,y,z)
GLfloat ctrlPoints[4][4][3]= {{{  -6.0f, -6.0f, 0.0f},     // u = 0,        v = 0
                               {   -6.0f, -2.0f, 0.0f},    //               v = 1
                               {   -6.0f,  2.0f, 0.0f},    //               v = 2
                               {   -6.0f,  6.0f, 0.0f}}, //                 v = 3

                              {{  -2.0f, -6.0f, 0.0f},     // u = 1,        v = 0
                               {   -2.0f, -2.0f, 8.0f},    //               v = 1
                               {   -2.0f,  2.0f, 8.0f},    //               v = 2
                               {   -2.0f,  6.0f, 0.0f }},//                 v = 3

                              {{  2.0f, -6.0f, 0.0f},      // u = 2,        v = 0
                                 {   -2.0f, -2.0f, 0.0f}, //                v = 1
```

```
{   -2.0f,   2.0f,  0.0f}, //                v = 2
{   -2.0f,   6.0f,  0.0f}},//                v = 3

{{   6.0f, -6.0f,  0.0f}, // u = 3,          v = 0
 {    6.0f, -2.0f,  0.0f},//                 v = 1
 {    6.0f,  2.0f,  0.0f},//                 v = 2
 {    6.0f,  6.0f,  0.0f}}};//               v = 3
```

```
// Knot sequence for the NURB
GLfloat Knots[8] = {0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, 1.0f};
```

图11-10 NURBS曲面

③剪裁

剪裁就是从NURBS曲面上切掉一部分，如图11-11所示。其中的NUBRS曲面与图11-10所示相同，只是切掉了一个三角形区域。把程序11-3加入到上面例子程序中就可以得到剪裁效果。即在gluBeginSurface/glEndSurface之间，由gluBeginTrim函数开始定义一个由gluPwlCurve给出的分段剪裁曲线，并由gluEndTrim结束这条曲线的定义。

程序 11-3

```
// Outside trimming points to include entire surface
GLfloat outsidePts[5][2] = /* counter clockwise */
        {{0.0f, 0.0f}, {1.0f, 0.0f}, {1.0f, 1.0f}, {0.0f, 1.0f}, {0.0f, 0.0f}};

// Inside trimming points to create triangle shaped hole in surface
GLfloat insidePts[4][2] = /* clockwise */
    {{0.25f, 0.25f}, {0.5f, 0.5f}, {0.75f, 0.25f}, { 0.25f, 0.25f}};
.....
        // Render the NURB
```

```
        // Begin the NURB definition
    gluBeginSurface(pNurb);

// Evaluate the surface
    gluNurbsSurface(pNurb,        // pointer to NURBS renderer
        8, Knots,                       // No. of knots and knot array u direction
        8, Knots,                       // No. of knots and knot array v direction
        4 * 3,                          // Distance between control points in u dir.
        3,                                      // Distance between control points in v dir.
        &ctrlPoints[0][0][0], // Control points
        4, 4,                                   // u and v order of surface
        GL_MAP2_VERTEX_3);              // Type of surface

        // Outer area, include entire curve
    gluBeginTrim (pNurb);
    gluPwlCurve (pNurb, 5, &outsidePts[0][0], 2, GLU_MAP1_TRIM_2);
    gluEndTrim (pNurb);

        // Inner triangluar area
    gluBeginTrim (pNurb);
        gluPwlCurve (pNurb, 4, &insidePts[0][0], 2, GLU_MAP1_TRIM_2);
    gluEndTrim (pNurb);

        // Done with surface
        gluEndSurface(pNurb);
```

图 11-11　剪裁 NURBS 曲面

在gluBeginTrim/gluEndTrim之间可以定义多个封闭的分段曲线。也可以由gluNurbsCurve函数定义剪裁曲线或该曲线的一部分，但是这些剪裁曲线的取值范围应该在(0.0，1.0)之间。

值得注意的是，剪裁曲线切掉在曲线右侧部分的曲面，即被定义为顺时针方向的剪裁曲线切掉其围成的内部区域，如图11-12所示。

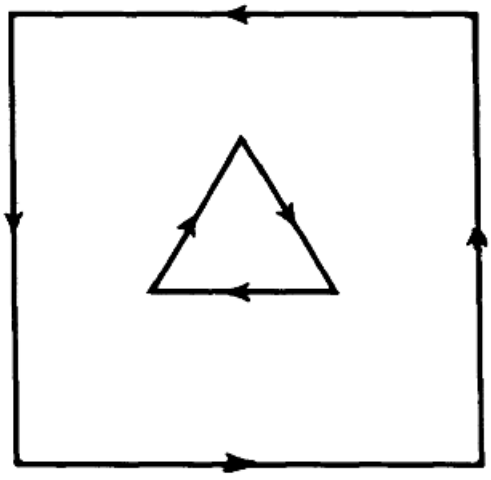


图11-12　剪裁曲线示意图

11.4　多边形镶嵌

GLU库(glu32.lib)提供了一个用于绘制复杂多边形和曲面的多边形镶嵌接口。在计算机图形学中，镶嵌就是把一组或多组点相连并形成一系列用于描述几何形状的填充多边形。多边形镶嵌的作用就是绘制有特殊属性的多边形（如绘制孔）。

1．复杂多边形

在OpenGL中，所谓的复杂多边形就是非凸多边形或有孔的多边形，如图11-13所示，而OpenGL的GL_POLYGON几何要素仅能绘制出简单的凸多边形。

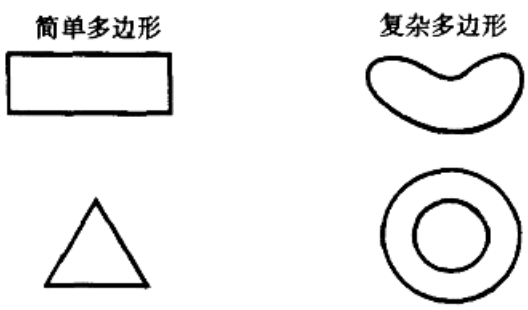


图11-13　复杂多边形

2．绘制凹多边形

绘制凹多边形的步骤如下：

(1) 创建一个镶嵌物体。

```
GLUtriangulatorObj     *tess;
tess=gluNewTess();
```

其中：GLUtriangulatorObj结构存贮用于绘制镶嵌多边形的状态信息。

(2) 调用一系列gluBeginPolygon，gluTessVertex和gluEndPolygon函数绘制多边形。

```
GLdouble vertics[100][3]

gluBeginPolygon(tess);
  gluTessVertex(tess, vertics[0], NULL);
  gluTessVertex(tess, vertics[1], NULL);
    ......
  gluTessVertex(tess, vertics[99], NULL);
gluEndPolygon(tess);
```

在调用gluEndPolygon函数后，开始进行镶嵌操作并生成一系列三角形。带状或扇形排列的三角形，由于这一过程要很长时间才能完成，为改进性能，最好把它存贮到显示列表中。

3. 绘制复杂多边形

绘制有孔或缠绕的多边形过程要比绘制凹多边形复杂一些。gluNextContour函数标识所定义路径的类型。表11-3给出gluNextContour的路径类型。

表11-3 gluNextContour的路径类型

路径类型	解　释
GLU_EXTERIOR	路径在多边形外面
GLU_INTERIOR	路径在多边形内部（孔）
GLU_UNKNOWN	未知的路径类型，由库函数辨别
GLU_CCW	把逆时针方向路径定义为外部路径，顺时针为内部路径（仅能使用一次）
GLU_CW	把顺时针方向路径定义为外部路径，逆时针为内部路径（仅能使用一次）

程序11-4定义一个外部轮廓和一个在中间的三角形内部路径，如图11-14所示。

程序11-4

```
#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */

#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLfloat rotation = 0.0;

/*
```

```
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK
reshape_scene(GLsizei width,      /* I - Width of the window in pixels */
              GLsizei height)     /* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing the letter "A".
 */

void CALLBACK
draw_scene(void)
{
  GLUtriangulatorObj     *tess;
  static GLdouble        outside[7][3] =
  {
    { 0.0, 1.0, 0.0 },
    { -0.5, -1.0, 0.0 },
    { -0.4, -1.0, 0.0 },
    { -0.2, -0.1, 0.0 },
    { 0.2, -0.1, 0.0 },
    { 0.4, -1.0, 0.0 },
    { 0.5, -1.0, 0.0 }
  };
  static GLdouble        inside[3][3] =
  {
    { 0.0, 0.6, 0.0 },
```

```
    { -0.1,  0.1,  0.0 },
    { 0.1,  0.1,  0.0 }
  };
  static float   red_light[4] = { 1.0,  0.0,  0.0,  1.0 };
  static float   red_pos[4] = { 1.0,  0.0,  0.0,  0.0 };
  static float   blue_light[4] = { 0.0,  0.0,  1.0,  1.0 };
  static float   blue_pos[4] = { -1.0,  0.0,  0.0,  0.0 };

 /*
  * Enable drawing features that we need...
  */

  glEnable(GL_DEPTH_TEST);
  glDisable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);

  glShadeModel(GL_SMOOTH);

 /*
  * Clear the color and depth buffers...
  */

  glClearColor(0.0,  0.0,  0.0,  0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 /*
  * Draw the cube and sphere in different colors...
  *
  * We have positioned two lights in this scene.  The first is red and
  * located above, to the right, and behind the viewer.  The second is blue
  * and located below, to the left, and in front of the viewer.
  */

  glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
  glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

  glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
  glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

  glEnable(GL_COLOR_MATERIAL);
```

```
 glPushMatrix();
   glTranslatef(0.0, 0.0, -15.0);
   glRotatef(-rotation, 0.0, 1.0, 0.0);

   glColor3f(0.0, 1.0, 0.0);
   glNormal3f(0.0, 0.0, 1.0);

   tess = gluNewTess();
   gluTessCallback(tess, GLU_BEGIN, glBegin);
   gluTessCallback(tess, GLU_VERTEX, glVertex3dv);
   gluTessCallback(tess, GLU_END, glEnd);

   gluBeginPolygon(tess);
     gluTessVertex(tess, outside[0], outside[0]);
     gluTessVertex(tess, outside[1], outside[1]);
     gluTessVertex(tess, outside[2], outside[2]);
     gluTessVertex(tess, outside[3], outside[3]);
     gluTessVertex(tess, outside[4], outside[4]);
     gluTessVertex(tess, outside[5], outside[5]);
     gluTessVertex(tess, outside[6], outside[6]);

   gluNextContour(tess, GLU_INTERIOR);
     gluTessVertex(tess, inside[0], inside[0]);
     gluTessVertex(tess, inside[1], inside[1]);
     gluTessVertex(tess, inside[2], inside[2]);
   gluEndPolygon(tess);
   gluDeleteTess(tess);

 glPopMatrix();

 auxSwapBuffers();
}

/*
 * 'rotate_objects()' - Rotate while we are idle...
 */

void CALLBACK
rotate_objects(void)
{
```

```
  rotation += 2.0;
  if (rotation >= 360.0)
    rotation -= 360.0;

  draw_scene();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void main(void)
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH);
  auxInitWindow("GLU Polygon Letter");

  auxReshapeFunc(reshape_scene);
  auxIdleFunc(rotate_objects);

  auxMainLoop(draw_scene);
}
```

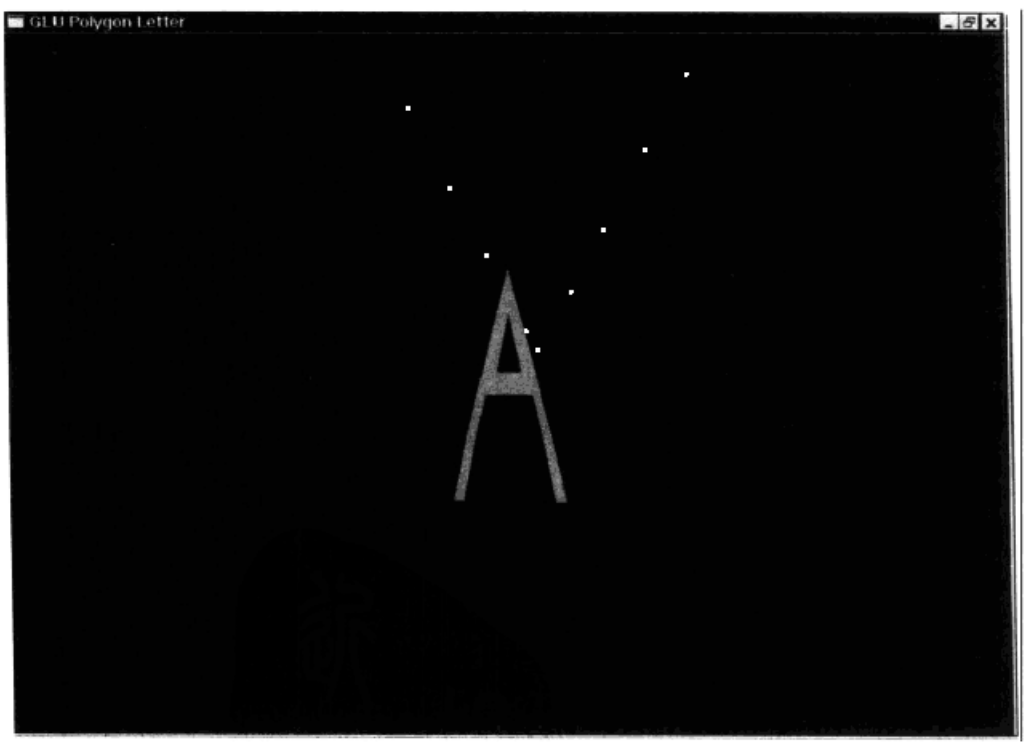


图11-14　多边形路径

4. 回调函数

glu库定义了几个用于获得特殊镶嵌效果的回调函数。gluTessCallback函数可以按用户的需要改变这些函数。

```
void gluTessCallback(GLUtrianglatorObj tobj, GLenum which, void (fn)());
```

回调函数必须在表11-4所列的范围。其中，GLU_BEGIN，GLU_END，GLU_VERTEX分别与glBegin，glEnd，glVertex3dv函数相对应。

下面的程序段用于显示镶嵌过程中出现的错误。

```
void tess_error_callback(GLenum error)
{
    MessageBeep(MB_ICONEXCLAMATION);
    MessageBox(NULL, gluErrorString(error), "GLU Error", MB_OK|
            MB_ICONEXCLAMATION);
}
```

表11-4 镶嵌回调函数

参 数	解 释
GLU_BEGIN	定义绘制GL_TRIANGLES, GL_TRANGLE_STRIP, GL_TRIANGLE_FAN的函数，该函数接受一个指定要绘制几何要素的枚举型参数，通常为glBegin
GLU_EDGE_FLAG	定义标识后面的GLU_VERTEX回调的函数，对于原有的顶点，这个函数接受GL_TRUE参数，对生成的顶点，接受GL_FALSE参数
GLU_VERTEX	定义发送每个顶点前调用的函数，通常为glVertex3dv，该函数接受gluTessVertex函数的第三个参量的拷贝
GLU_END	定义标识绘制几何要素完成的函数，通常为glEnd，无参量
GLU_ERROR	定义标识出现错误时调用的函数，接受一个枚举型参数

11.5 相关函数

```
void gluQuadricNormals (GLUquadricObj *quadricObject, GLenum normals);
void gluQuadricTexture (GLUquadricObj *quadricObject, GLenum textureCoords);
void gluQuadricOrientation (GLUquadricObj *quadricObject, GLenum orientation);
void gluQuadricDrawStyle (GLUquadricObj *quadricObject, GLenum drawStyle);
GLUQuadricObj* gluNewQuadric (void);
void gluDeleteQuadric (GLUquadricObj *state);
void gluQuadricCallback (GLUquadricObj *qobj, GLenum which, void (*fn)());
void gluBeginPolygon (GLUtriangulatorObj *tobj);
void gluEndPolygon (GLUtriangulatorObj *tobj);
void gluNextContour (GLUtriangulatorObj *tobj);
void gluTessVertex (GLUtriangulatorObj *tobj, GLdouble v[3], void *data);
GLUtriangulatorObj* gluNewTess (void);
void gluTessCallback (GLUtriangulatorObj *tobj, GLenum which, void (*fn)());
void gluDeleteTess (GLUtriangulatorObj *tobj);
void gluCylinder (GLUquadricObj *qobj, GLdouble baseRadius, GLdouble topRadius, GLdouble
```

height, GLint slices, GLint stacks);

void gluDisk (GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops);

void gluPartialDisk (GLUquadricObj *qobj, GLdouble innerRadius, GLdouble outerRadius, GLint slices, GLint loops, GLdouble startAngle, GLdouble sweepAngle);

void gluSphere (GLUquadricObj *qobj, GLdouble radius, GLint slices, GLint stacks);

GLUnurbsObj* gluNewNurbsRender (void);

void gluDeleteNurbsRender (GLUnurbsObj *nobj);

void gluNurbsCallback (GLUnurbsObj *nobj, GLenum which, void (*fn)());

void gluBeginCurve (GLUnurbsObj *nobj);

void gluEndCurve (GLUnurbsObj *nobj);

void gluNurbsCurve (GLUnurbsObj *nobj, GLint nknots, GLfloat *knot, GLint stride, GLfloat *ctlarray, GLint order, GLenum type);

void gluBeginSurface (GLUnurbsObj *nobj);

void gluEndSurface (GLUnurbsObj *nobj);

void gluNurbsSurface (GLUnurbsObj *nobj, GLint uknot_count, GLfloat *uknot, GLint vknot_count, GLfloat *vknot, GLint u_stride, GLint v_stride, GLfloat *ctlarray, GLint sorder, GLint vorder, GLenum type);

void gluBeginTrim (GLUnurbsObj *nobj);

void gluEndTrim (GLUnurbsObj *nobj);

void gluPwlNurbs (GLUnurbsObj *nobj, GLint count, GLfloat *array, GLint stride, GLenum type);

void gluLoadSamplingMatrices (GLUnurbsObj *nobj, GLfloat modelMatrix[16], const GLfloat projMatrix[16], const GLint viewport[4]);

void gluNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat value);

void gluGetNurbsProperty (GLUnurbsObj *nobj, GLenum property, GLfloat *value);

void glMapGrid1{fd}(GLint n, TYPE u1, TYPE u2);

void glMapGrid2{fd}(GLint un, TYPE u1, TYPE u2, GLint vn, TYPE v1, TYPE v2);

void glEvalMesh1(GLenum mode, GLint i1, GLint i2);

void glEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);

void glEvalPoint1(GLint i);

void glEvalPoint2(GLint i, GLint j);

void glEvalCoord1{fd}{v}(TYPE u);

void glEvalCoord2{fd}{v}(TYPE u, TYPE v);

void glGetMap{idf}{v}(GLenum target, GLenum query, TYPE *v);

void glMap1{fd}(GLenum target, TYPE u1, TYPE u2, GLint stride, GLint order, const TYPE *points);

void glMap2{fd}(GLenum target, TYPE u1, TYPE u2, GLint ustride, GLint uorder, TYPE v1, TYPE v2, GLint vstride, GLint vorder, const TYPE *points);

void glEvalPoint1(GLint i);

void glEvalPoint2(GLint i, GLint j);

void glGetMap{idf}{v}(GLenum target, GLenum query, TYPE *v);

第十二章　使用缓冲区

有效地使用缓冲区是 OpenGL 的一个重要特征，缓冲区的作用不仅局限在使用双缓冲技术制作动画和利用深度缓冲区进行曲面消隐等方面，还有其他方面的用途。本章将详细讲述这些缓冲区的使用方法。

OpenGL 提供如下几种与图形相关的缓冲区，每一缓冲区都有特定的功能：

- 颜色缓冲区
- 深度缓冲区
- 模板缓冲区
- 累加缓冲区

12.1　什么是缓冲区?

OpenGL 的缓冲区是与窗口上或脱离屏幕的图像相对应的一个二维数组。每一缓冲区的行、列与当前窗口绘图区相对应，但取值范围和数据类型不同。

使用 OpenGL 之前，应该为需要的颜色模式和缓冲区配置硬件设备描述表(HDC)。这些信息存贮在 PIXELFORMATDESCRIPTOR 结构中。设置缓冲区的方式如下：

```
// This structure holds buffer, layer and color mode information
PIXELFORMATDESCRIPTOR pfd;

// first initialize the pfd size and version…
pfd.nSize=sizeof(pfd);
pfd.Version=1;

// Next layer and buffer information…
pfd.dwFlags=PFD_DRAW_TO_WINDOWS|PFD_SUPPORT_OPENGL
pfd.dwLayerMask=PFD_MAIN_PLANE;
pfd.dwLayerType=PFD_MAIN_PLANE;

// The pixel type indicate weather we use color indices or RGBA
pfd.iPixelType=PFD_TYPE_RGBA;

// Now we specify the minimum number of bitplanes we need for each buffer.
// windows will choose the closest pixel format satisfying our min. requirement
pfd.cColorBits=8;
pfd.cDepthbits=16;
```

```
pfd.cAccumbits=0;
pfd.cStencilbits=0;
```

dwFlags 表示在窗口中用 OpenGL 绘图，同时也指定所需颜色缓冲区的数目，如表 12-1 所示。

表 12-1 PIXELFORMATDESCRIPTOR 的标识符

标 识 符	解 释
PFD_DRAW_TO_WINDOW	在窗口中绘图
PFD_DRAW_TO_BITMAP	在脱离屏幕的位图上完成绘图
PFD_SUPPORT_GDI	颜色缓冲区支持 GDI 命令
PFD_SUPPORT_OPENGL	缓冲区支持 OpenGL 绘图命令
PFD_DOUBLEBUFFER	颜色值采用双缓冲技术
PFD_STEREO	可以使用两套缓冲区（前、后缓冲区）
PFD_DOUBLE_BUFFER_DONTCARE	是否使用双缓冲都不产生影响
PFD_STEREO_DONTCARE	是否为立体缓冲区都不产生影响

dwLayerMask 和 iLayerType 指定要使用的绘图平面，通常设置成 PFD_MAIN_PLANE。有些 OpenGL 图形卡除了具有常规 Windows 色彩平面外还有辅助缓冲区，用于在不覆盖主图像的情况下，绘制菜单或其他图形。

iPixelType 指定颜色值的类型，为 PFD_TYPE_RGBA 或 PFD_TYPE_COLORINDEX 之一。

cColorBits，cDepthBits，cAccumBits 和 cStencilBits 分别指定窗口的各个缓冲区大小。若 cColorBits 为零，Windows 使用最小彩色模式(4 位、8 位)。如其他字位为零，则不使用所对应的缓冲区。若 iPixelType 为 PFD_TYPE_RGBA，cColorBits 指定 RGB 颜色总的字位数目。Microsoft 的 OpenGL 实现不支持 α 字位。

在给定 PIXELFORMATDESCRIPTOR 结构的信息后，可以用下面的调用设置窗口的像素格式。

```
// The device context refers to the graphics driver for this window.
HDC hdc;
// This integer holds the window pixel format code
int pf;

// Choose and select the pixel format
pf=ChoosePixelFormat(hdc, &pfd);
if (pf==0)
{
    // Could not find the pixel format
    MessageBox(NULL, " ChoosePixelFormat failed!", "Error", MB_OK);
}
else if (!SetPixelFormat(hdc, &pfd))
{
    // Could not set the pixel format
    MessageBox(NULL, " SetPixelFormat failed!", "Error", MB_OK);
}
```

调用 ChoosePixelFormat 函数后，用选定的硬件值给 PIXELFORMATDESCRIPTOR 结构赋

值。返回时，dwFlags 字位有另外三个标识位需要引起重视，如表 12-2 所示。

表 12-2 PIXELFORMATDESCRIPTOR 的返回值

返 回 值	解 释
PFD_GENERIC_FORMAT	OpenGL 基本实现所支持的格式
PFD_NEED_PALETTE	RGBA 颜色缓冲区要在由调色板管理的设备上绘图，需要逻辑调色板
PFD_NEED_SYSTEM_PALETTE	需要固定的系统调色板，才能正确显示颜色值，需要调用 SetSystemPalette 函数设置逻辑调色板与系统调色板的一一映射关系

若 PFD_NEED_PALETT 已赋值，则要用 cRedBits，cGreenBits，cBlueBits 一一定义一个逻辑调色板。下面是一个定义调色板的例子。

```
HDC hdc;
PIXELFORMATDESCRIPTOR pfd;
HPALETTE palette;
LOGPALETTE *pal;
int i, pf, num_colors,
    red, num_reds,
    green, num_greens;
    blue, num_blues;

// Get the current pixel format informat
pf=GetPixelFormat(hdc);
DescribePixelFormat(hdc, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

// Check to see if we need to make a palette
if (pfd.dwgFlags & PFD_NEED_PALETTE)
{
    // Yes, we do. First, allocate logical color palette entries
    num_colors=1<<pfd.cColorBits;
    pal=(PLOGPALETTE)localAlloc(LMEM_FIXED, sizeof(LOGPALETTE)+
                                +numcolors*sizeof(PALETTEENTRY));
    pal->palVersion=0x300;
    pal->palNumEntries=num_colors;

    num_reds=(1<<pfd.cRedBits)-1;
    num_greens=(1<<pfd.cGreenBits)-1;
    num_blues=(1<<pfd.cBlueBits)-1;

    for(blue=0, i=0; blue<=num_blues); blue++)
        for(green=0, i=0; green<=num_greens); green++)
            for(red=0, i=0; red <=num_ red s); red ++)
            {
                pal->palPalEntry[i].peRed=255*red/num_reds;
```

```
                pal->palPalEntry[i].peGreen=255*red/num_greens;
                pal->palPalEntry[i].peBlue=255*red/num_Blues;
                pal->palPalEntry[i].peFlags=0;
            }
    palette=CreatePalette(pal);
    SelectPalette(hdc, palette, FALSE);
    RealizePalette(hdc);

    LocalFree(pal);
}
```

12.2 颜色缓冲区

颜色缓冲区存贮像素颜色信息，像素的颜色可以是一个色彩指数，也可以是一个 RGBA 值。在屏幕上显示 RGBA 像素的颜色直接用 RGB 最接近的颜色，而色彩指数像素要由颜色查对表的指数得到一个颜色。在 Windows 环境下，这个查对表用逻辑色彩调色表实现。

1. 双缓冲区技术

双缓冲区技术提供一个脱离屏幕的颜色缓冲区，常用于动画制作。使用双缓冲区时，可以通过把一个脱屏的场景迅速交换到屏幕上，以消除动画的闪烁。应该注意的是，所谓双缓冲区仅仅针对颜色缓冲区而言，是不存在第二个深度、模板、累加缓冲区的。在把像素格式设为双缓冲区时，OpenGL 选择在后缓冲区中绘图。可以用 glDrawBuffer 函数指定表 12-3 所列的值。

表 12-3 glDrawBuffer 指定的值

缓 冲 区	解 释
GL_FRONT	仅在前颜色缓冲区（可见）绘图
GL_BACK	仅在后颜色缓冲区（不可见）绘图
GL_FRONT_AND_BACK	在前、后颜色缓冲区绘图

2. 立体缓冲区技术

立体缓冲区技术为单缓冲区模式另外提供一个颜色缓冲区，为双缓冲区模式再提供两个颜色缓冲区，从而生成左视、右视屏幕图像。正确选择取景位置，可以生成真正三维图像。通常要把左视、右视图像分开一定距离以模拟人的两瞳孔间距。在这一模式下，glDrawBuffer 可选值如表 12-4 所示。在绝大多数 PC 机显示卡上不支持这一模式。

表 12-4 立体缓冲区

缓 冲 区	解 释
GL_LEFT_FRONT	仅在前左颜色缓冲区绘图
GL_LEFT_BACK	仅在后左颜色缓冲区绘图
GL_RIGHT_FRONT	仅在前右颜色缓冲区绘图
GL_RIGHT_BACK	仅在后右颜色缓冲区绘图
GL_FRONT	仅在前左、前右颜色缓冲区绘图
GL_BACK	仅在后左、后右颜色缓冲区绘图

3. 交换缓冲区

在 Windows 环境下，调用

```
SwapBuffers(hdc);
```

可以交换前、后缓冲区的内容。其中 hdc 为正在绘图窗口的设备描述表，若选择了立体缓冲像素格式，进行上述调用同时交换左视、右视图像。

12.3 深度缓冲区

深度缓冲区存贮每一个像素的距离。这个距离为缩放到当前剪裁体积的 near/far 参数值后，像素与观察者的距离。windows 环境下，软件实现的 OpenGL 支持 16 位和 32 位深度值。

深度缓冲区通常用于曲面消隐操作，选择不同的深度检验函数可以得到两种相反的效果：GL_GREATER 用于消除物体被遮挡的部分；GL_LESS 则除掉能遮住其他物体的部分，与前者现实的视觉效果相反，如图 12-1 所示。

1. 深度比较

在窗口中用 OpenGL 绘图时，像素的 z 值要与深度缓冲区存贮的值相比较。若通过了比较，则把该像素同深度值一起存贮到颜色缓冲区中。OpenGL 定义了 8 个比较函数用于深度检验，如表 12-5 所示。缺省的比较函数为 GL_LESS。调用 glDepthFunc 函数，可以选择比较函数。

```
glDepthFunc (fuction);
```

表 12-5 深度检验函数

函数名称	解 释
GL_NEVER	不会通过
GL_LESS	z 值小于缓冲区的值时，通过
GL_EQUAL	z 值等于缓冲区的值时，通过
GL_LEQUAL	z 值不大于缓冲区的值时，通过
GL_NOTEQUAL	z 值不等于缓冲区的值时，通过
GL_GREATER	z 值大于缓冲区的值时，通过
GL_GEQUAL	z 值不小于缓冲区的值时，通过
GL_ALWAYS	全部通过

2. 深度值

在使用 GL_EQUAL 和 GL_NOTEQUAL 深度比较函数时，有时要用 glDepthRange 函数减少可用值的数目。

```
glDepthRange(near, far);
```

near，far 参数为[0.0,1.0]区间的一个浮点数。near，far 缺省值分别为 0.0，1.0。通常 near<far，在使用 GL_GREATER 或 GL_GEQUAL 比较函数时，需要把两者关系颠倒过来。值得注意的是，减少深度缓冲区的取值范围不影响剪裁，但在曲面消隐时，可能会由于深度缓冲区不精确而出错。

有时，深度比较需要一个不同的初值。缺省时，调用 glClear 函数后，深度缓冲区的值为 1.0。glClearDepth 函数可以指定一个不同的初值。

```
glClearDepth(depth);
```

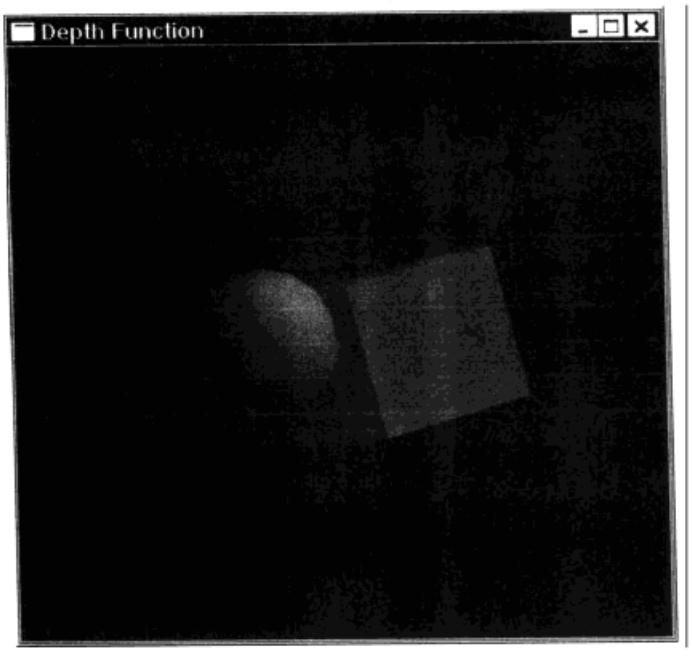


图 12-1a　检验函数 GL_LESS

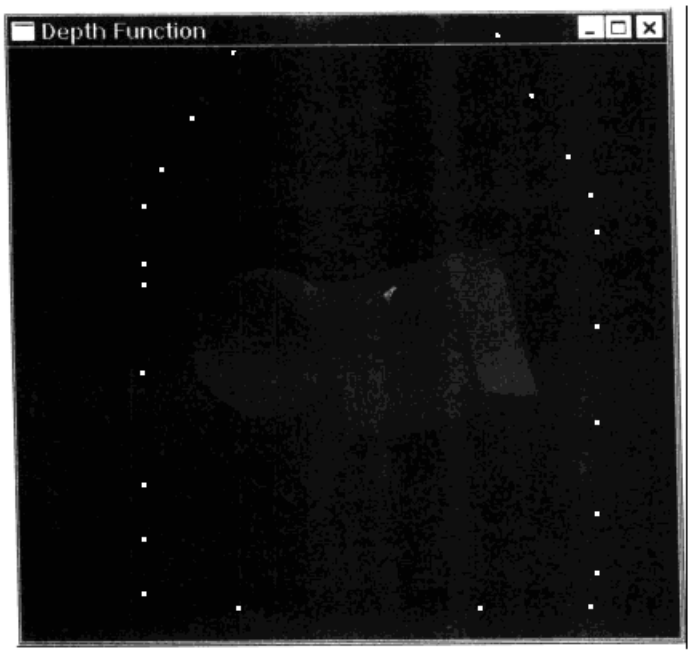


图 12-1b　检验函数 GL_GREATER

如果没有调用 glDepthRange 函数，把深度缓冲区的 near，far 参数设置成更小范围，depth 为[0.0，1.0]区间内的一个浮点数。通常，在使用 GL_GREATER，GL_GEQUAL 比较函数时，depth 为 0.0，GL_LESS 和 GL_LEQUAL 比较函数对应的 depth 值为 1.0。

3．深度缓冲区的应用

深度缓冲区通常用于曲面消隐。程序 12-1 为这一应用的演示程序，显示的图形如图 12-1 所示。程序中，比较了 GL_LESS 和 GL_GREATER 两个比较函数的差别，在采用这两个比较函数时，分别用 glClearDepth 函数把深度缓冲区的初值设为 1.0 和 0.0，但是选用不正确的

初值时，显示的图像不会正确。

程序 12-1

```
/*
 * Press the 'd' key to toggle between GL_LESS and GL_GREATER depth
 * tests.  Press the 'ESC' key to quit.
 */

#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */

#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLenum    depth_function = GL_LESS;     /* Current depth function */

/*
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK
reshape_scene(GLsizei width,  /* I - Width of the window in pixels */
              GLsizei height)/* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);
```

```
  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */

void CALLBACK
draw_scene(void)
{
  static float red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
  static float red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
  static float blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
  static float blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };

 /*
  * Enable drawing features that we need...
  */

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);

  glShadeModel(GL_SMOOTH);
  glDepthFunc(depth_function);

 /*
  * Clear the color and depth buffers...
  */

  if (depth_function == GL_LESS)
    glClearDepth(1.0);
  else
    glClearDepth(0.0);

  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  /*
```

```
 * Draw the cube and sphere in different colors...
 *
 * We have positioned two lights in this scene.  The first is red and
 * located above, to the right, and behind the viewer.  The second is blue
 * and located below, to the left, and in front of the viewer.
 */

 glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
 glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

 glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
 glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

 glPushMatrix();
   glTranslatef(-1.0, 0.0, -20.0);
   auxSolidSphere(1.0);
 glPopMatrix();

 glPushMatrix();
   glTranslatef(1.0, 0.0, -20.0);
   glRotatef(15.0, 0.0, 1.0, 0.0);
   glRotatef(15.0, 0.0, 0.0, 1.0);
   auxSolidCube(2.0);
 glPopMatrix();

 glFlush();
}

/*
 * 'toggle_depth()' - Toggle the depth function between GL_LESS and GL_GREATER.
 */

void CALLBACK
toggle_depth(void)
{
  if (depth_function == GL_LESS)
    depth_function = GL_GREATER;
  else
    depth_function = GL_LESS;
}
/*
```

```
* 'main()' - Initialize the window and display the scene until the user presses
*            the ESCape key.
*/

void
main(void)
{
  auxInitDisplayMode(AUX_RGB | AUX_SINGLE | AUX_DEPTH);
  auxInitWindow("Depth Function");
  auxKeyFunc(AUX_d, toggle_depth);
  auxReshapeFunc(reshape_scene);
  auxMainLoop(draw_scene);
}
```

4. 另一个应用实例

深度缓冲区也可以用来生成场景的等值线映射，即每一个深度值用一种颜色表示。调用glRealPixels函数可以实现这一功能。

```
glRealPixels(x, y, width, height, GL_DEPTH_COMPONENT, type, pixels);
```

对返回的深度值进行缩放并给它赋予一个颜色值，可以用于显示等值线。上述功能在色彩指数模式下更容易实现。例如：

```
#define WIDTH 320
#define HEIGHt 200
GLfloat pixels[WIDTH*HEIGHT];
int i;

// Draw the scene
glEnable(GL_DEPTH_TEST);
.....
// Grab the depth buffer
glReadPixel(0, 0, WIDTH, HEIGHT, GL_DEPTH_COMPONENT, GL_FLOAT, pixels);
// Convert depth values to color indices
for (i=0; i<(WIDTH*HEIGHT); i++)
     pixels[i]=pixels[i]*255.0; //Assume 256 color palette
// Display the new pixels on the screen
gldisable(GL_DEPTH_TEST);
glDrawPixels(0, 0, WIDTH, HEIGHT, GL_COLOR_INDEX, GL_FLOAT, pixels);
```

在实际应用中，使用 RGBA 颜色值，可以更好地控制调色板和颜色的范围，以便得到更好的场景效果。这种情况下，可以调用 glBlendFunc 函数，把图像与所得到的深度图像混合起来。详见本书第十三章关于混合操作的内容。

5. 切掉一部分场景

下面举一个切掉一部分场景的例子（见程序 12-2），这个程序的核心是 draw_scene 函数，该函数用一个运动平面切开一个方块和圆球。首先，绘制运动平面，但不在颜色缓冲区

内绘图。

```
    glDrawBuffer(GL_NONE);

 glColor3i(0, 0, 0);
 glBegin(GL_POLYGON);
   glVertex3f(-100.0, 100.0, cutting_plane);
   glVertex3f(100.0, 100.0, cutting_plane);
   glVertex3f(100.0, -100.0, cutting_plane);
   glVertex3f(-100.0, -100.0, cutting_plane);
 glEnd();

 glDrawBuffer(GL_BACK);
```

运动平面是一个宽度、高度均为200的正方形，其深度值的取值范围在[-30.0，-15.0]之间，运动平面的定义为：

```
void CALLBACK move_plane(void)
{
  cutting_plane += cutting_dir;

 /*
  * Reverse directions as needed...
  */

  if (cutting_plane <= -30.0 ||
      cutting_plane >= -15.0)
    cutting_dir = -cutting_dir;
```

画完平面后，再允许使用颜色缓冲区用来绘制方块和圆球。这样，才能有效地完成切掉部分场景的操作。

程序 12-2

```
* Press the 'd' key to toggle between GL_LESS and GL_GREATER depth
 * tests.  Press the 'ESC' key to quit.
 */

#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */
```

```
#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLenum    depth_function = GL_LESS;     /* Current depth function */
GLfloat   cutting_plane = -15.0,        /* Cutting plane distance */
     cutting_dir    = -1.0;         /* Cutting plane direction */

/*
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK reshape_scene(GLsizei width,  /* I - Width of the window in pixels */
              GLsizei height)/* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */

void CALLBACK
draw_scene(void)
{
  static float red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
  static float red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
  static float blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
  static float blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };
```

```
/*
 * Enable drawing features that we need...
 */

 glEnable(GL_DEPTH_TEST);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glEnable(GL_LIGHT1);

 glShadeModel(GL_SMOOTH);
 glDepthFunc(depth_function);

/*
 * Clear the color and depth buffers...
 */

 if (depth_function == GL_LESS)
   glClearDepth(1.0);
 else
   glClearDepth(0.0);

 glClearColor(0.0, 0.0, 0.0, 0.0);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*
 * Draw the cutting plane.  Note that we disable drawing into the normal color buffer
 * while we do this...
 */

 glDrawBuffer(GL_NONE);

 glColor3i(0, 0, 0);
 glBegin(GL_POLYGON);
   glVertex3f(-100.0, 100.0, cutting_plane);
   glVertex3f(100.0, 100.0, cutting_plane);
   glVertex3f(100.0, -100.0, cutting_plane);
   glVertex3f(-100.0, -100.0, cutting_plane);
 glEnd();

 glDrawBuffer(GL_BACK);
```

```
/*
 * Draw the cube and sphere in different colors...
 *
 * We have positioned two lights in this scene.  The first is red and
 * located above, to the right, and behind the viewer.  The second is blue
 * and located below, to the left, and in front of the viewer.
 */

 glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
 glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

 glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
 glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

 glPushMatrix();
   glTranslatef(-1.0, 0.0, -20.0);
   auxSolidSphere(1.0);
 glPopMatrix();

 glPushMatrix();
   glTranslatef(1.0, 0.0, -20.0);
   glRotatef(15.0, 0.0, 1.0, 0.0);
   glRotatef(15.0, 0.0, 0.0, 1.0);
   auxSolidCube(2.0);
 glPopMatrix();

 auxSwapBuffers();
}

/*
 * 'toggle_depth()' - Toggle the depth function between GL_LESS and GL_GREATER.
 */

void CALLBACK toggle_depth(void)
{
  if (depth_function == GL_LESS)
    depth_function = GL_GREATER;
  else
    depth_function = GL_LESS;
}
```

```
/*
 * 'move_plane()' - Move the cutting plane while we are idle...
 */

void CALLBACK move_plane(void)
{
  cutting_plane += cutting_dir;

 /*
  * Reverse directions as needed...
  */

  if (cutting_plane <= -30.0 ||
      cutting_plane >= -15.0)
    cutting_dir = -cutting_dir;

  draw_scene();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void main()
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH);
  auxInitWindow("Depth Function");

  auxKeyFunc(AUX_d, toggle_depth);
  auxReshapeFunc(reshape_scene);
  auxIdleFunc(move_plane);

  auxMainLoop(draw_scene);
}
```

12.4 模板缓冲区

OpenGL 的模板缓冲区提供了更多控制屏幕上图形的方法以及许多深度缓冲区无法完成的功能。一个最简单的应用就是用模板缓冲区遮住屏幕上的某一区域，也可以用模板缓冲区

创建光源照射物体形成的阴影。根据不同的图形硬件，可以生成来自多个光源形成的软、硬阴影，使得到的场景更逼真。

1. 使用模板缓冲区

在 Windows 环境下，使用模板缓冲区之前，要为窗口的像素格式描述符设置 cStencilBits，即

```
pfd.cStencilBits=1;
```

其后，要调用 glEnable (GL_STENCIL_TEST)函数，允许使用模板缓冲区。

2. 模板缓冲区函数

在 OpenGL 中有四个模板函数，即

```
void glClearStencil(GLint s);
void glStencilFunc(Glenum func, GLint ref, GLuint mask);
void glStencilMask(GLuint mask);
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

glClearStencil 函数与 glClearColor，glClearDepth 等函数相似，在调用 glClear(GL_STENCLL_BIT)后，为模板缓冲区提供一个初值，缺省值为 0。与颜色缓冲区、深度缓冲区不同的是，不必每次重画场景都消除模板缓冲区。

3. 在模板缓冲区中绘图

在完成上面的步骤后，还要设置对模板缓冲区的操作方式。缺省时，不对模板缓冲区做任何操作，可以在窗口的任意位置绘图。为使模板工作，需要调用 glStencilFunc 和 glStencilOp 函数。

glStencilFunc 函数定义一个比较函数、参考值和用于模板缓冲区操作的位屏蔽。这些函数有效值见表 12-6。

表 12-6 模板检验函数

函数名称	解 释
GL_NEVER	不会通过
GL_LESS	z 值<ref 时，通过
GL_EQUAL	z 值=ref 时，通过
GL_LEQUAL	z 值≤ref 时，通过
GL_NOTEQUAL	z 值≠ref 时，通过
GL_GREATER	z 值> ref 时，通过
GL_GEQUAL	z 值≥ref 时，通过
GL_ALWAYS	全部通过

与模板函数相关的是模板操作，由 glStencilOp 函数定义。有效的模板操作见表 12-7。

表 12-7 模板操作

模板操作	解 释
GL_KEEP	保留当前值不变
GL_ZERO	把模板缓冲区的值置为零
GL_REPLACE	把模板缓冲区的值置为 glStencilFunc 定义的 ref
GL_INCR	把模板缓冲区的值加 1，限制有意义的最大无符号值(即小于)
GL_DECR	把模板缓冲区的值减 1，限制在零(即大于)
GL_INVERT	把模板缓冲区的值字位颠倒

通常，位屏蔽图像给出绘图区域的轮廓。下面为把位屏蔽图像写入模板缓冲区的例子。

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

然后，在模板缓冲区中存贮数值 1。在绘制场景前，用模板缓冲区位屏蔽绘图。

```
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

程序 12-3 为用模板缓冲区切掉方块中间部分的例子。应该注意的是，在这个程序中，并没有使用深度缓冲区，而是利用模板缓冲区得到了这个效果。下面是程序的核心：

```
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

glPushMatrix();
  glTranslatef(-1.0, 0.0, -20.0);
  auxSolidSphere(1.0);
glPopMatrix();
```

在绘制模板图像后，在没有圆球的位置上绘制方块，如图 12-2 所示。

程序 12-3

```
#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */

#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

/*
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK
reshape_scene(GLsizei width,  /* I - Width of the window in pixels */
              GLsizei height)/* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */
```

```
  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */

void CALLBACK
draw_scene(void)
{
  static float red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
  static float red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
  static float blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
  static float blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };

 /*
  * Enable drawing features that we need...
  */

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_STENCIL_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);

  glShadeModel(GL_SMOOTH);

 /*
  * Clear the color, depth, and stencil buffers...
  */

  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
```

```
        GL_STENCIL_BUFFER_BIT);

/*
 * Draw the sphere that will be cutting away parts of the cube...
 */

 glStencilFunc(GL_ALWAYS, 1, 1);
 glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);

 glPushMatrix();
   glTranslatef(-1.0, 0.0, -20.0);
   auxSolidSphere(1.0);
 glPopMatrix();

/*
 * Clear the color and depth buffers once again...
 */

 glClearColor(0.0, 0.0, 0.0, 0.0);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*
 * Draw the cube...
 *
 * We have positioned two lights in this scene.  The first is red and
 * located above, to the right, and behind the viewer.  The second is blue
 * and located below, to the left, and in front of the viewer.
 */

 glStencilFunc(GL_NOTEQUAL, 1, 1);
 glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

 glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
 glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

 glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
 glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

 glPushMatrix();
   glTranslatef(1.0, 0.0, -20.0);
   glRotatef(15.0, 0.0, 1.0, 0.0);
```

```
    glRotatef(15.0, 0.0, 0.0, 1.0);
    auxSolidCube(2.0);
  glPopMatrix();

  auxSwapBuffers();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void main()
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH | AUX_STENCIL);
  auxInitWindow("Stenciling");

  auxReshapeFunc(reshape_scene);
  auxMainLoop(draw_scene);
}
```

图 12-2　模板缓冲区的例子

12.5　累加缓冲区

累加缓冲区支持运动模糊、景深等特殊效果，也支持全屏幕反走样。与本章介绍的前面几个缓冲区相比，累加缓冲区比较简单，只有一个相关函数 glAccum 管理所有对累加缓冲

区的操作。这些操作见表 12-8。

表 12-8 累加缓冲区操作

缓冲区操作	解　　释
GL_ACCUM	经过缩放的 RGBA 值加到累加缓冲区
GL_LOAD	读入 RGBA 值,存入相应的累加缓冲区单元,覆盖当前值
GL_ADD	给累加缓冲区的每个 RGBA 值加上 value
GL_MULT	给累加缓冲区的每个 RGBA 值乘上 value,并写入在累加缓冲区的位置
GL_RETURN	把累加缓冲区的值传给颜色缓冲区或当前选定要写入的缓冲区

通常，使用累加缓冲区的方法是在其中绘制多个视图，并用 glAccum(GL_RETURN, 1.0) 返回合成后的场景。

1．运动模糊

绘制运动模糊视觉效果要比绘制图像帧的序列并在两帧间移动照相机要复杂得多，因为只有人的视觉无法跟踪高速运动物件时，才感觉到所谓的运动模糊。实际上，大脑处理图像时，这些图片仍在运动，但一直聚焦在目标上。

模拟运动模糊效果时，要使物体当前位置的图像比其他位置更清楚一些。实现这一效果要赋给当前位置的图像较大的颜色放大因子。实现的程序如下：

```
/* Draw the current frame */
draw_frame();
/* Load the accumulation buffer with 50% of the current frame */
glAccum(GL_LOAD, 0.5);

/* Draw the last 10 frames and accumulate 5% for each */
for (i=1; i<=10; i++)
{
     draw_frame(-i);
     glAccum(GL_LOAD, 0.05);
}
/* display the final scene */
glAccum(GL_RETURN, 1.0);
```

值得注意的是，开始时，不用 glClear 函数初始化累加缓冲区，而是对场景的第一帧用 glAccum(GL_LOAD,S)。程序 12-4 为对方块和圆球做运动模糊处理的演示程序。

程序 12-4

```
#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */
```

```
#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLfloat rotation = 0.0;

/*
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK
reshape_scene(GLsizei width,  /* I - Width of the window in pixels */
              GLsizei height) /* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */

void CALLBACK
draw_scene(void)
{
  GLfloat       frame;
  static float red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
  static float red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
  static float blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
  static float blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };
```

```
/*
 * Enable drawing features that we need...
 */

 glEnable(GL_DEPTH_TEST);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glEnable(GL_LIGHT1);

 glShadeModel(GL_SMOOTH);

/*
 * Clear the color and depth buffers...
 */

 glClearColor(0.0, 0.0, 0.0, 0.0);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*
 * Draw the cube and sphere in different colors...
 *
 * We have positioned two lights in this scene.  The first is red and
 * located above, to the right, and behind the viewer.  The second is blue
 * and located below, to the left, and in front of the viewer.
 */

 glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
 glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

 glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
 glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

/*
 * Draw the objects 11 times starting at the current rotation...
 */

 for (frame = 0.0; frame <= 11.0; frame ++)
 {
   glPushMatrix();
     glTranslatef(0.0, 0.0, -20.0);
     glRotatef(rotation - frame, 0.0, 1.0, 0.0);
```

```
    glPushMatrix();
      glTranslatef(-1.0, 0.0, 0.0);
      auxSolidSphere(1.0);
    glPopMatrix();

    glPushMatrix();
      glTranslatef(1.0, 0.0, 0.0);
      glRotatef(15.0, 0.0, 1.0, 0.0);
      glRotatef(15.0, 0.0, 0.0, 1.0);
      auxSolidCube(2.0);
    glPopMatrix();
  glPopMatrix();

  /*
   * Accumulate 50% the first time, 5% every other time...
   */

  if (frame == 0.0)
    glAccum(GL_LOAD, 0.5);
  else
    glAccum(GL_ACCUM, 0.05);
 };

 /*
  * Copy the accumulated results back to the color buffer...
  */

 glAccum(GL_RETURN, 1.0);

 auxSwapBuffers();
}

/*
 * 'rotate_objects()' - Rotate while we are idle...
 */

void CALLBACK
rotate_objects(void)
{
  rotation += 2.0;
  if (rotation >= 360.0)
```

```
    rotation -= 360.0;

  draw_scene();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void main()
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH | AUX_ACCUM);
  auxInitWindow("Motion Blur");

  auxReshapeFunc(reshape_scene);
  auxIdleFunc(rotate_objects);

  auxMainLoop(draw_scene);
}
```

2. 反走样

累加缓冲区的另一个应用是进行全场景反走样处理。所谓反走样就是把图像在几个方向上都跳动 1/2 像素，使图像边缘模糊，例如，把 4 个跳动的场景累加到一起得到的图像非常光滑。

用累加缓冲区做反走样处理要牺牲一些绘图速度。如果做实时反走样动画，就要选用支持多取样的图形硬件做反走样操作，而用累加缓冲区的速度太慢了。

如果处理静止图像，累加缓冲区不仅可以做反走样，还可以模拟景深效果，这是多取样不能完成的。有关内容请参见《精通 OpenGL》一书。

12.6 相关函数

```
void glAccum(GLenum op, GLfloat value);
void glClearAccum(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);
void glClearDepth(GLclampd depth);
void glClearIndex(GLfloat c);
void glClearstencil(GLint s);
void glDepthFunc (GLenum func);
void glDepthMask (GLboolean flag);
void glDepthRange (GLclampd rear, GLclampd far);
```

第十三章　视觉效果：混合、雾

本章讲述 OpenGL 提供的颜色混和及雾函数，用这些函数得到的视觉效果会使屏幕显示的图像更加生动、逼真。

颜色混合函数支持半透明效果，可以用于模拟窗户、玻璃杯和其他透明物体。雾函数在所绘制的多边形上添加某种颜色，使场景变得朦胧。在 8 位显示卡上模拟这两种视觉效果是不太理想的。

13.1　混　合

OpenGL 提供的混合函数可以直接控制存贮在颜色缓冲区中的像素 RGBA 值，但是在色彩指数模式下不能做混合操作。调用混合函数前，要先调用 glEnable(GL_BLEND)，允许使用颜色混合功能。然后，调用 glBlendFunc 指定混合函数的两参数：源颜色和目标颜色，参见表 13-1，表 13-2。这两个参数的缺省值分别为 GL_ONE 和 GL_ZERO，相当于不使用混合操作，即调用了 glDisable(GL_BLEND)。

表 13-1　源颜色参数

参　数	解　释
GL_ZERO	源颜色为(0, 0, 0, 0)
GL_ONE	使用源颜色
GL_DST_COLOR	源颜色与目标像素颜色相乘
GL_ONE_MINUS_DST_COLOR	源颜色与((1, 1, 1, 1)-目标颜色)相乘
GL_SRC_ALPHA	源颜色与源颜色的 α 组分相乘
GL_DST_ALPHA	源颜色与目标颜色的 α 组分相乘，Microsoft OpenGL 不支持
GL_ONE_MINUS_DST_ALPHA	源颜色与((1, 1, 1, 1)-目标颜色的 α 组分)相乘，Microsoft OpenGL 不支持
GL_SRC_ALPHA_STATURE	源颜色与 min{源颜色的 α 组分,((1, 1, 1, 1)-目标颜色的 α 组分)}相乘，Microsoft OpenGL 不支持

13.1.1　半透明效果

半透明效果是混合操作的一个典型应用，用于窗户、水杯等可以透视的三维物体。半透明效果也可以把多个图像组合起来，或在画笔程序中做“笔刷”。

在上面提到的应用中，做混合操作都要调用下面的函数：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

把源颜色乘以 RGBA 颜色的 α 值再加上目标像素颜色乘以(1-α)，即混合函数把当前绘图颜色的一部分覆盖到屏幕的像素上。若 α =0，为全透明，α =1，为完全不透明。混合操作可以表述成：

$R_d=R_s*A_s+R_d*(1-A_s)$

$G_d=G_s*A_s+G_d*(1-A_s)$

$B_d=B_s*A_s+B_d*(1-A_s)$

其中，下标“s”表示源颜色，下标“d”表示目标颜色。值得注意的是，由于常规的深度检验会影响半透明的效果，在绘制多边形时，要按照由后向前的顺序进行。

表 13-2 目标颜色参数

参 数	混 合 因 子
GL_ZERO	目标颜色为(0, 0, 0, 0)
GL_ONE	使用目标颜色
GL_SRC_COLOR	目标颜色与源像素颜色相乘
GL_ONE_MINUS_SRC_COLOR	目标颜色与((1, 1, 1, 1)-源颜色)相乘
GL_SRC_ALPHA	目标颜色与源颜色的α组分相乘
GL_DST_ALPHA	目标颜色与目标颜色的α组分相乘，Microsoft OpenGL 不支持
GL_ONE_MINUS_SRC_ALPHA	目标颜色与((1, 1, 1, 1)-源颜色的α组分)相乘
GL_ONE_MINUS_DST_ALPHA	目标颜色与((1, 1, 1, 1)-目标颜色的α组分)相乘，Microsoft OpenGL 不支持
GL_SRC_ALPHA_STATURE	目标颜色与 min{源颜色的α组分,((1, 1, 1, 1)-目标颜色的α组分)}相乘，Microsoft OpenGL 不支持

程序 13-1 用于绘制图 13-1 所示的茶壶。在 draw_scene 函数中，由后向前绘制两个茶壶，使观察者可以透过前面的茶壶看到后面的茶壶。图 13-1 显示的效果不够理想，可以先做深度检验并调用 glEnable(GL_CULL_FACE)，剔除朝后面的多边形加以改进。

程序 13-1

```
#include <GL/glaux.h>

/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */

#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLfloat rotation = 0.0;

/*
 * 'reshape_scene()' - Change the size of the scene...
 */
```

图 13-1　半透明的茶壶

```
void CALLBACK
reshape_scene(GLsizei width,     /* I - Width of the window in pixels */
              GLsizei height)    /* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */

void CALLBACK
draw_scene(void)
{
  GLfloat       frame;
```

```
static float  red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
static float  red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
static float  blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
static float  blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };

/*
 * Enable drawing features that we need...
 */

 glEnable(GL_DEPTH_TEST);
 glEnable(GL_LIGHTING);
 glEnable(GL_LIGHT0);
 glEnable(GL_LIGHT1);

 glShadeModel(GL_SMOOTH);

/*
 * Clear the color and depth buffers...
 */

 glClearColor(0.0, 0.0, 0.0, 0.0);
 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

/*
 * Draw the cube and sphere in different colors...
 *
 * We have positioned two lights in this scene.  The first is red and
 * located above, to the right, and behind the viewer.  The second is blue
 * and located below, to the left, and in front of the viewer.
 */

  glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
  glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

  glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
  glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

  glEnable(GL_COLOR_MATERIAL);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

  glPushMatrix();
```

```
    glTranslatef(0.0, 0.0, -15.0);
    glRotatef(-rotation, 0.0, 1.0, 0.0);

    glDisable(GL_BLEND);
    glColor3f(1.0, 1.0, 0.0);
    auxSolidTeapot(1.0);
  glPopMatrix();

  glPushMatrix();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rotation, 0.0, 1.0, 0.0);

    glEnable(GL_BLEND);
    glColor4f(1.0, 1.0, 1.0, 0.25);
    auxSolidTeapot(1.0);
  glPopMatrix();

  auxSwapBuffers();
}

/*
 * 'rotate_objects()' - Rotate while we are idle...
 */

void CALLBACK
rotate_objects(void)
{
  rotation += 2.0;
  if (rotation >= 360.0)
    rotation -= 360.0;

  draw_scene();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void
main(void)
```

```
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH);
  auxInitWindow("Blended Teapot");

  auxReshapeFunc(reshape_scene);
  auxIdleFunc(rotate_objects);

  auxMainLoop(draw_scene);
}
```

在 draw_scene 函数中，应该注意以下几点：

(1) 按源颜色的 α 值做混合操作。

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

(2) 绘制后面不透明的茶壶时，应该关闭 OpenGL 的混合功能。

```
glDisable(GL_BLEND);
glColor3f(1.0, 1.0, 0.0);
auxSolidTeapot(1.0);
```

(3) 绘制前面的茶壶时，先需允许使用混合操作并把 α 设为 0.25。

```
glEnable(GL_BLEND);
glColor4f(1.0, 1.0, 1.0, 0.25);
auxSolidTeapot(1.0);
```

13.1.2 反走样操作

同样，可以用混合操作的参数 GL_SRC_ALPHA 和 GL_ONE_MINUS_SRC_ALPHA 强化反走样点、线、多边形的效果。在图形硬件支持反走样和混合效果的系统上，混合操作与(在累加缓冲区中进行的)全屏幕反走样作用相似，但是混合操作在屏幕的绘图一次完成，所以要比累加操作的速度快好几倍。

在场景中用混合操作绘制反走样几何要素的函数调用如下：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POINT_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);
```

13.1.3 一个画笔程序

在画笔程序中，用混合函数创建一个画笔。首先，要定义画笔的 α 图像。所谓的 α 图像是 RGBA 颜色中只有 α 组分，而没有 RGB 组合。α 图像可以定义在画布上涂颜色的多少，如图 13-2 所示。用画笔绘画时，调用的混合函数如下：

```
glBlendFunc(GL_SRC_COLOR, GL_ONE_MINUS_SRC_ALPHA);
```

即用当前颜色绘图，故画笔所用颜色为：

$R_d=R_s*A_s+R_d*(1-A_s)$

$G_d=G_s*A_s+G_d*(1-A_s)$

$B_d=B_s*A_s+B_d*(1-A_s)$

程序 13-2 是一个用 7×7 像素的笔刷画图的程序。按住鼠标器的左键，则在鼠标当前位置调用 DrawXY 函数绘图。

```
glRasterPos2(mousex, mousey);
glDrawPixels(7, 7, GL_LUMINANCE_ALPHA,, GL_UNSIGNED_BYTE, BlendBrush[0]);
```

RepaintWindow 函数在窗口大小发生变化等情况下，重新绘制窗口。

```
glViewport(0, 0, rect->right, rect->bottom);
glOrtho(0.0, (float)rect->right, (float)rect->bottom, 0.0, -1.0, 1.0);

glClearColor(0.0, 0.0, 0.0, 1.0);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

程序 13-2

```
/*
 *   WinMain()              - Main entry...
 *   DisplayErrorMessage()  - Display an error message dialog.
 *   MakePalette()          - Make a color palette for RGB colors if necessary.
 *   BlendProc()            - Handle window events in the viewing window.
 *   RepaintWindow()        - Redraw the client area.
 *   DrawXY()               - Draw in the client area.
 *
*/

/*
 * Include necessary headers.
 */

#include <windows.h>
#include <GL/gl.h>
#include "blendraw.h"
#include <stdarg.h>
#include <math.h>
#ifndef M_PI
#  define M_PI (double)3.14159265358979323846
```

```
#endif /* !M_PI */

/*
 * Globals...
 */

HWND      BlendWindow;        /* Blend window */
HPALETTE  BlendPalette;       /* Color palette (if necessary) */
HDC       BlendDC;        /* Drawing context */
HGLRC          BlendRC;       /* OpenGL rendering context */

unsigned char  BlendBrush[7][16] =
{
{ 0xff, 0x00, 0xff, 0x00, 0xff, 0x08, 0xff, 0x10, 0xff, 0x08, 0xff, 0x00, 0xff, 0x00 },
  { 0xff, 0x00, 0xff, 0x08, 0xff, 0x10, 0xff, 0x20, 0xff, 0x10, 0xff, 0x08, 0xff, 0x00 },
  { 0xff, 0x08, 0xff, 0x10, 0xff, 0x20, 0xff, 0x40, 0xff, 0x20, 0xff, 0x10, 0xff, 0x08 },
  { 0xff, 0x10, 0xff, 0x20, 0xff, 0x40, 0xff, 0x80, 0xff, 0x40, 0xff, 0x20, 0xff, 0x10 },
  { 0xff, 0x08, 0xff, 0x10, 0xff, 0x20, 0xff, 0x40, 0xff, 0x20, 0xff, 0x10, 0xff, 0x08 },
  { 0xff, 0x00, 0xff, 0x08, 0xff, 0x10, 0xff, 0x20, 0xff, 0x10, 0xff, 0x08, 0xff, 0x00 },
  { 0xff, 0x00, 0xff, 0x00, 0xff, 0x08, 0xff, 0x10, 0xff, 0x08, 0xff, 0x00, 0xff, 0x00 },
};

GLboolean Drawing = GL_FALSE; /* GL_TRUE if drawing */

/*
 * Local functions...
 */

void          DisplayErrorMessage(char *, ...);
void          MakePalette(int);
LRESULT CALLBACK   BlendProc(HWND, UINT, WPARAM, LPARAM);
void          DrawXY(int, int);
void          RepaintWindow(RECT *);

/*
 * 'WinMain()' - Main entry...
 */

int APIENTRY
WinMain(HINSTANCE hInst,      /* I - Current process instance */
     HINSTANCE hPrevInstance, /* I - Parent process instance */
```

```
    LPSTR     lpCmdLine,          /* I - Command-line arguments */
    int       nCmdShow)       /* I - Show window at startup? */
{
  MSG         msg;            /* Window UI event */
  WNDCLASS    wc;             /* Window class */
  POINT       pos;            /* Current mouse pos */

 /*
  * Register main window...
  */

  wc.style         = 0;
  wc.lpfnWndProc   = (WNDPROC)BlendProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInst;
  wc.hIcon         = NULL;
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = 0;
  wc.lpszMenuName  = MAKEINTRESOURCE(IDR_MENU1);
  wc.lpszClassName = "Blend Paint";

  if (RegisterClass(&wc) == 0)
  {
    DisplayErrorMessage("Unable to register window class!");
    return (FALSE);
  };

 /*
  * Then create it...
  */

  BlendWindow = CreateWindow("Blend Paint", "Blend Paint",
                             WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,
                  32, 32, 400, 300,
                  NULL, NULL, hInst, NULL);

  if (BlendWindow == NULL)
  {
    DisplayErrorMessage("Unable to create window!");
    return (FALSE);
```

```
  };

  ShowWindow(BlendWindow, nCmdShow);
  UpdateWindow(BlendWindow);

 /*
  * Loop on events until the user quits this application...
  */

  while (GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  };

  return (msg.wParam);
}

/*
 * 'DisplayErrorMessage()' - Display an error message dialog.
 */

void
DisplayErrorMessage(char *format, /* I - printf() style format string */
                    ...)      /* I - Other arguments as necessary */
{
  va_list ap;             /* Argument pointer */
  char        s[1024];        /* Output string */

  if (format == NULL)
    return;

  va_start(ap, format);
  vsprintf(s, format, ap);
  va_end(ap);

  MessageBeep(MB_ICONEXCLAMATION);
  MessageBox(NULL, s, "Error", MB_OK | MB_ICONEXCLAMATION);
}

/*
```

```
* 'MakePalette()' - Make a color palette for RGB colors if necessary.
*/

void
MakePalette(int pf)           /* I - Pixel format ID */
{
  PIXELFORMATDESCRIPTOR pfd;      /* Pixel format information */
  LOGPALETTE        *pPal;        /* Pointer to logical palette */
  int               nColors;  /* Number of entries in palette */
  int               i,        /* Color index */
                    rmax,         /* Maximum red value */
                    gmax,         /* Maximum green value */
                    bmax;         /* Maximum blue value */

 /*
  * Find out if we need to define a color palette...
  */

  DescribePixelFormat(BlendDC, pf, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

  if (!(pfd.dwFlags & PFD_NEED_PALETTE))
  {
    BlendPalette = NULL;
    return;
  };

 /*
  * Allocate memory for a color palette...
  */

  nColors = 1 << pfd.cColorBits;

  pPal = (LOGPALETTE *)malloc(sizeof(LOGPALETTE) +
                              nColors * sizeof(PALETTEENTRY));

  pPal->palVersion    = 0x300;
  pPal->palNumEntries = nColors;

 /*
  * Get the maximum values for red, green, and blue.  Then build 'nColors'
  * colors...
```

```
 */

 rmax = (1 << pfd.cRedBits) - 1;
 gmax = (1 << pfd.cGreenBits) - 1;
 bmax = (1 << pfd.cBlueBits) - 1;

 for (i = 0; i < nColors; i ++)
 {
   pPal->palPalEntry[i].peRed   = 255 * ((i >> pfd.cRedShift) & rmax) / rmax;
   pPal->palPalEntry[i].peGreen = 255 * ((i >> pfd.cGreenShift) & gmax) / gmax;
   pPal->palPalEntry[i].peBlue  = 255 * ((i >> pfd.cBlueShift) & bmax) / bmax;

   pPal->palPalEntry[i].peFlags = 0;
 };

/*
 * Create, select, and realize the palette...
 */

 BlendPalette = CreatePalette(pPal);
 SelectPalette(BlendDC, BlendPalette, FALSE);
 RealizePalette(BlendDC);

 free(pPal);
}

/*
 * 'BlendProc()' - Handle window events in the viewing window.
 */

LRESULT CALLBACK
BlendProc(HWND   hWnd,        /* I - Window triggering this event */
      UINT   uMsg,        /* I - Message type */
      WPARAM wParam,          /* I - 'word' parameter value */
      LPARAM lParam)          /* I - 'long' parameter value */
{
  int              pf;  /* Pixel format ID */
  PIXELFORMATDESCRIPTOR pfd; /* Pixel format information */
  PAINTSTRUCT      ps;  /* WM_PAINT message info */
  RECT             rect;     /* Current client area rectangle */
```

```
switch (uMsg)
{
  case WM_CREATE :
     /*
      * 'Create' message.  Get device and rendering contexts, and
   * setup the client area for OpenGL drawing...
   */

   BlendDC = GetDC(hWnd);

      pfd.nSize        = sizeof(pfd);
      pfd.nVersion     = 1;
      pfd.dwFlags      = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL;
                           /* Do OpenGL drawing */
      pfd.dwLayerMask  = PFD_MAIN_PLANE;  /* Main drawing plane */
      pfd.iPixelType   = PFD_TYPE_RGBA;   /* RGB color buffer */
      pfd.cColorBits   = 0;          /* Best color buffer please */
      pfd.cDepthBits   = 0;          /* Don't need a depth buffer */
      pfd.cStencilBits = 0;          /* No stencil buffer */
      pfd.cAccumBits   = 0;          /* No accumulation buffer */

      pf = ChoosePixelFormat(BlendDC, &pfd);
    if (pf == 0)
        DisplayErrorMessage("texscene was unable to choose a suitable pixel format!");
      else if (!SetPixelFormat(BlendDC, pf, &pfd))
        DisplayErrorMessage("texscene was unable to set the pixel format!");

      MakePalette(pf);

      BlendRC = wglCreateContext(BlendDC);
      wglMakeCurrent(BlendDC, BlendRC);
    break;

   case WM_SIZE :
   case WM_PAINT :
      /*
       * Repaint the client area with our bitmap...
    */

    BeginPaint(hWnd, &ps);
```

```
    GetClientRect(hWnd, &rect);
  RepaintWindow(&rect);

  EndPaint(hWnd, &ps);
  break;

 case WM_COMMAND :
    /*
     * Handle menu selections...
  */

  switch (LOWORD(wParam))
  {
    case IDM_FILE_EXIT :
        DestroyWindow(BlendWindow);
        break;
  };
     break;

  case WM_QUIT :
  case WM_CLOSE :
     /*
      * Destroy the windows and bitmaps and exit...
      */

      DestroyWindow(BlendWindow);

   exit(0);
      break;

  case WM_DESTROY :
     /*
      * Release and free the device context, rendering
      * context, and color palette...
   */

   if (BlendRC)
     wglDeleteContext(BlendRC);

   if (BlendDC)
     ReleaseDC(BlendWindow, BlendDC);
```

```
if (BlendPalette)
  DeleteObject(BlendPalette);

   PostQuitMessage(0);
   break;

case WM_QUERYNEWPALETTE :
   /*
    * Realize the color palette if necessary...
*/

    if (BlendPalette)
{
      SelectPalette(BlendDC, BlendPalette, FALSE);
      RealizePalette(BlendDC);

      InvalidateRect(hWnd, NULL, FALSE);
  return (TRUE);
};
    break;

case WM_PALETTECHANGED:
   /*
    * Reselect our color palette if necessary...
*/

 if (BlendPalette && (HWND)wParam != hWnd)
 {
      SelectPalette(BlendDC, BlendPalette, FALSE);
      RealizePalette(BlendDC);

      UpdateColors(BlendDC);
 };
 break;

 case WM_LBUTTONDOWN :     /* Left button = red */
    Drawing = GL_TRUE;
 glColorMask(GL_TRUE, GL_FALSE, GL_FALSE, GL_TRUE);
 DrawXY(LOWORD(lParam), HIWORD(lParam));
 break;
```

```
    case WM_MBUTTONDOWN :      /* Middle button = green */
        Drawing = GL_TRUE;
     glColorMask(GL_FALSE, GL_TRUE, GL_FALSE, GL_TRUE);
     DrawXY(LOWORD(lParam), HIWORD(lParam));
     break;

    case WM_RBUTTONDOWN :      /* Right button = blue */
        Drawing = GL_TRUE;
     glColorMask(GL_FALSE, GL_FALSE, GL_TRUE, GL_TRUE);
     DrawXY(LOWORD(lParam), HIWORD(lParam));
     break;

    case WM_MOUSEMOVE :
        if (Drawing)
       DrawXY(LOWORD(lParam), HIWORD(lParam));
        break;

    case WM_LBUTTONUP :
    case WM_MBUTTONUP :
    case WM_RBUTTONUP :
        Drawing = GL_FALSE;
     break;

    default :
       /*
        * Pass all other messages through the default window
        * procedure...
        */
        return (DefWindowProc(hWnd, uMsg, wParam, lParam));
  };

  return (FALSE);
}

/*
 * 'DrawXY()' - Draw at the given mouse position.
 */

void
DrawXY(int mousex,        /* I - Horizontal mouse position */
       int mousey)        /* I - Vertical mouse position */
{
```

```
  glRasterPos2i(mousex, mousey);
  glDrawPixels(7, 7, GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, BlendBrush[0]);

  glFinish();
}
/*
 * 'RepaintWindow()' - Redraw the client area.
 */

void
RepaintWindow(RECT *rect)     /* I - Client area rectangle */
{
 /*
  * Reset the viewport and clear the window to white...
  */

  glViewport(0, 0, rect->right, rect->bottom);
  glOrtho(0.0, (float)rect->right, (float)rect->bottom, 0.0, -1.0, 1.0);

  glClearColor(0.0, 0.0, 0.0, 1.0);
  glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
  glClear(GL_COLOR_BUFFER_BIT);

  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

  glFinish();
}
```

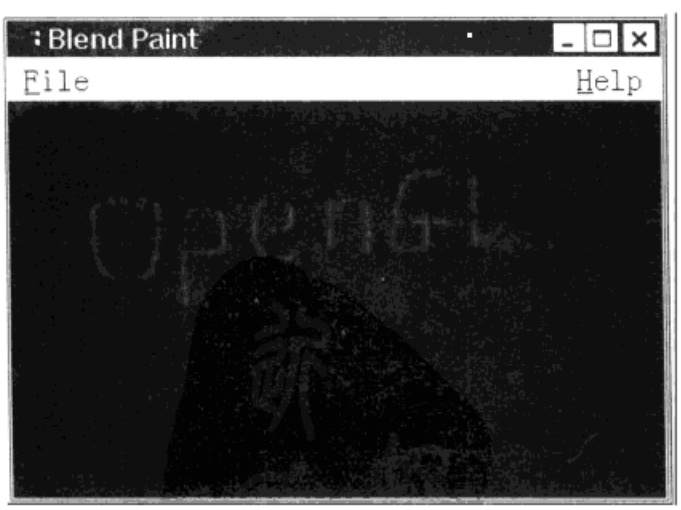


图 13-2　画笔程序

13.2 雾效果

通过 OpenGL 的 glFog 函数可以实现“深度暗示”(由距离决定颜色的明暗)和空气效果。实现雾效果就是依据每一个顶点与观察者的距离，给纹理图像添加一个预定义的颜色。

OpenGL 支持三种雾效果：GL_LINEAR 用于“深度暗示”，如图 13-3 所示；GL_EXP 用于模拟浓雾或云；GL_EXP2 用于烟或薄雾。分别用下面的函数调用选择这三种雾效果。

```
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogi(GL_FOG_MODE, GL_EXP);
glFogi(GL_FOG_MODE, GL_EXP2);
```

在选定了雾类型后，还要调用 glFogfv 或 glFogiv 函数选择与场景混合的雾颜色。

```
GLfloat fog_color[4]={r, g, b, a};
glFogfv(GL_FOG_COLOR, fog_color);

GLint fog_color[4]={r, g, b, a};
glFogiv(GL_FOG_COLOR, fog_color);
```

对于深度暗示，通常选择的颜色与背景的颜色相同。这样，才能在视觉上得到远处物体的颜色更接近背景颜色的感觉。有时，也会选择一个比较亮的颜色，使物体在背景衬托下更显突出。

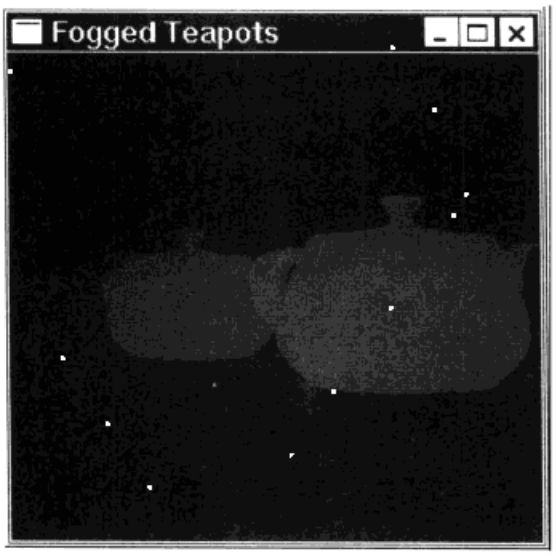


图 13-3 使用 glFog 函数

13.2.1 绘制深度暗示的茶壶

程序 13-3 绘制有深度暗示的两个茶壶。draw_scene 函数完成绘图功能，首先，把雾颜色选为黑色，雾的种类为 GL_LINEAR。

程序 13-3

```
#include <GL/glaux.h>
```

```
/*
 * These #define constants are provided for compatibility between MS Windows
 * and the rest of the world.
 *
 * CALLBACK and APIENTRY are function modifiers under MS Windows.
 */

#ifndef WIN32
#  define CALLBACK
#  define APIENTRY
#endif /* !WIN32 */

GLfloat rotation = 0.0;

/*
 * 'reshape_scene()' - Change the size of the scene...
 */

void CALLBACK
reshape_scene(GLsizei width,  /* I - Width of the window in pixels */
              GLsizei height) /* I - Height of the window in pixels */
{
 /*
  * Reset the current viewport and perspective transformation...
  */

  glViewport(0, 0, width, height);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(22.5, (float)width / (float)height, 0.1, 1000.0);

  glMatrixMode(GL_MODELVIEW);
}

/*
 * 'draw_scene()' - Draw a scene containing a cube with a sphere in front of
 *                  it.
 */
```

```
void CALLBACK
draw_scene(void)
{
  static float red_light[4] = { 1.0, 0.0, 0.0, 1.0 };
  static float red_pos[4] = { 1.0, 1.0, 1.0, 0.0 };
  static float blue_light[4] = { 0.0, 0.0, 1.0, 1.0 };
  static float blue_pos[4] = { -1.0, -1.0, -1.0, 0.0 };
  static float fog_color[4] = { 0.0, 0.0, 0.0, 0.0 };

 /*
  * Enable drawing features that we need...
  */

  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);

  glShadeModel(GL_SMOOTH);

 /*
  * Clear the color and depth buffers...
  */

  glClearColor(0.0, 0.0, 0.0, 0.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

 /*
  * Draw the cube and sphere in different colors...
  *
  * We have positioned two lights in this scene.  The first is red and
  * located above, to the right, and behind the viewer.  The second is blue
  * and located below, to the left, and in front of the viewer.
  */
  glLightfv(GL_LIGHT0, GL_DIFFUSE, red_light);
  glLightfv(GL_LIGHT0, GL_POSITION, red_pos);

  glLightfv(GL_LIGHT1, GL_DIFFUSE, blue_light);
  glLightfv(GL_LIGHT1, GL_POSITION, blue_pos);

  glEnable(GL_FOG);
```

```
  glFogf(GL_FOG_DENSITY, 0.05);
  glFogfv(GL_FOG_COLOR, fog_color);

  glPushMatrix();
    glTranslatef(-1.0, 0.0, -15.0);
    glRotatef(-rotation, 0.0, 1.0, 0.0);

    auxSolidTeapot(1.0);
  glPopMatrix();

  glPushMatrix();
    glTranslatef(1.0, 0.0, -10.0);
    glRotatef(rotation, 0.0, 1.0, 0.0);

    auxSolidTeapot(1.0);
  glPopMatrix();

  auxSwapBuffers();
}

/*
 * 'rotate_objects()' - Rotate while we are idle...
 */

void CALLBACK
rotate_objects(void)
{
  rotation += 2.0;
  if (rotation >= 360.0)
    rotation -= 360.0;

  draw_scene();
}

/*
 * 'main()' - Initialize the window and display the scene until the user presses
 *            the ESCape key.
 */

void
main(void)
```

```
{
  auxInitDisplayMode(AUX_RGB | AUX_DOUBLE | AUX_DEPTH);
  auxInitWindow("Fogged Teapots");

  auxReshapeFunc(reshape_scene);
  auxIdleFunc(rotate_objects);

  auxMainLoop(draw_scene);
}
```

使用其他类型的雾，可以把雾颜色选定为白色或其他淡颜色。对于 GL_EXP 和 GL_EXP2 类型雾效果，还要选择一个雾的浓度：

```
glFogf(GL_FOG_DENSITY, density);
```

通常 density 的取值应该大于 0.0，小于 1.0，图 13-4 所示为 density 系数对雾颜色的影响。

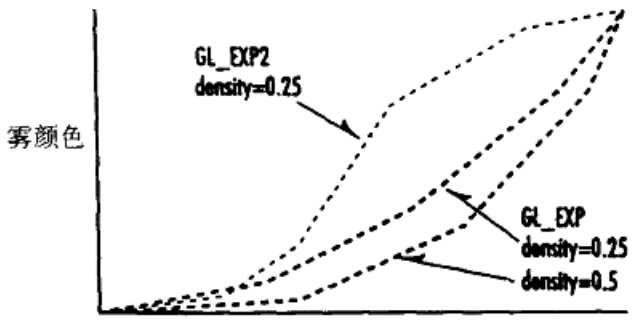


图 13-4　density 系数对雾颜色的影响

13.2.2　雾效果中的距离因素

雾的距离变换到 glVertex 调用的 z 分量，取值为 0.0～1.0 并存贮到深度缓冲区。雾的浓度和距离决定混合用的雾颜色，即

GL_EXP:　　$f=\exp-(density\times z)$

GL_EXP2:　　$f=\exp-(density\times z)^2$

GL_LINEAR:　$f=(end-z)/(end-start)$

缺省情况下，对深度为 0.0～1.0 的所有深度做雾操作。GL_FOG_START 和 GL_FOG_END 系列限定用于雾计算的深度范围。

13.3　相关函数

```
void glBlendFunc(GLenum sfactor, GLenum dfactor);
void glFog {fi} (GLenum pname, TYPE param);
void glFog {fi}v (GLenum pname, const TYPE *params);
```

第十四章　交互式图形程序设计

使用前面几章介绍 OpenGL 的功能可以创建复杂的三维场景，但是在很多应用程序中(如设计游戏程序)，除了菜单和对话框之外，还需要用户与场景的交互作用，在 Windows 环境下，这一交互作用通常用鼠标器完成。

作为 OpenGL 功能强大的一个特征，选择机制允许用户在窗口某一位置点击鼠标器来确定所指向的物体。选择屏幕上一特定物体的过程称为拾取。选择机制可以指定一个取景体积及其中的物体。OpenGL 函数根据屏幕坐标和指定的像素大小生成一个矩阵。用户可以用这个矩阵在鼠标指针处创建一个小取景体积，然后检验其包含的物体。

利用 OpenGL 的反馈机制，用户可以得到写入帧缓冲区时顶点变换、光照的信息。利用这一信息可以把显示的结果发送网络上、发给绘图仪或者把有交互功能的 OpenGL 场景加到 GDI 图形上。尽管反馈与选择机制的目的不同，但两者的操作方式基本相同。如果把这两个机制同时使用，功能会更强大。

14.1　选　择

实际上，选择是一种绘图模式，但并不把像素拷贝到帧缓冲区，而是使取景体积内的几何要素在选择缓冲区中生成命中记录。

使用选择缓冲区前，应对其进行设置并给几何要素或几何要素组命名，以便在选择缓冲区中能够正确辨别出来，然后分析选择缓冲区，确定与取景体积相交的物体，只有在进入选择模式前，修改取得体积并调用绘图程序才能确定在场景中所限定区域的物体，否则只能得到边缘上的值。

14.1.1　命名几何要素

通常并不是给场景中每一个几何要素都命名，而是给几组几何要素命名，从而给场景中特定的物体或物体的某一部分命名。与显示列表相似，物体名称也是无符号的整数。

名称列表存贮在名称堆栈中。名称堆栈经过初始化，就可以推入名称或替代堆栈最顶层的名称。在选择过程中，名称堆栈的所有名称都拷贝到选择缓冲区中，因此，必要时，一次命中可以返回多个名称。

作为一个简单例子，下面介绍太阳系行星的简化模型。按下鼠标器左键时，对话框就显示选中的行星。程序 14-1 为一段绘图程序。

程序 14-1

```
#define SUN         1
#define MERCURY     2
```

```
#define VENUS  3
#define EARTH  4
#define MARS   5

// Called to draw scene
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Save the matrix state and do the rotations
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    // Translate the whole scene out and into view
    glTranslatef(0.0f, 0.0f, -300.0f);

    // Initialize the names stack
    glInitNames();
    glPushName(0);

    // Set material color, Yellow
    // Sun
    glRGB(255, 255, 0);
    glLoadName(SUN);
    auxSolidSphere(15.0f);

    // Draw Mercury
    glRGB(128,0,0);
    glPushMatrix();
    glTranslatef(24.0f, 0.0f, 0.0f);
    glLoadName(MERCURY);
    auxSolidSphere(2.0f);
    glPopMatrix();

    // Draw Venus
    glPushMatrix();
    glRGB(128,128,255);
    glTranslatef(60.0f, 0.0f, 0.0f);
    glLoadName(VENUS);
    auxSolidSphere(4.0f);
```

```
glPopMatrix();

// Draw the Earth
glPushMatrix();
glRGB(0,0,255);
glTranslatef(100.0f,0.0f,0.0f);
glLoadName(EARTH);
auxSolidSphere(8.0f);
glPopMatrix();

// Draw Mars
glRGB(255,0,0);
glPushMatrix();
glTranslatef(150.0f, 0.0f, 0.0f);
glLoadName(MARS);
auxSolidSphere(4.0f);
glPopMatrix();

// Restore the matrix state
glPopMatrix(); // Modelview matrix

// Flush drawing commands
glFlush();
}
```

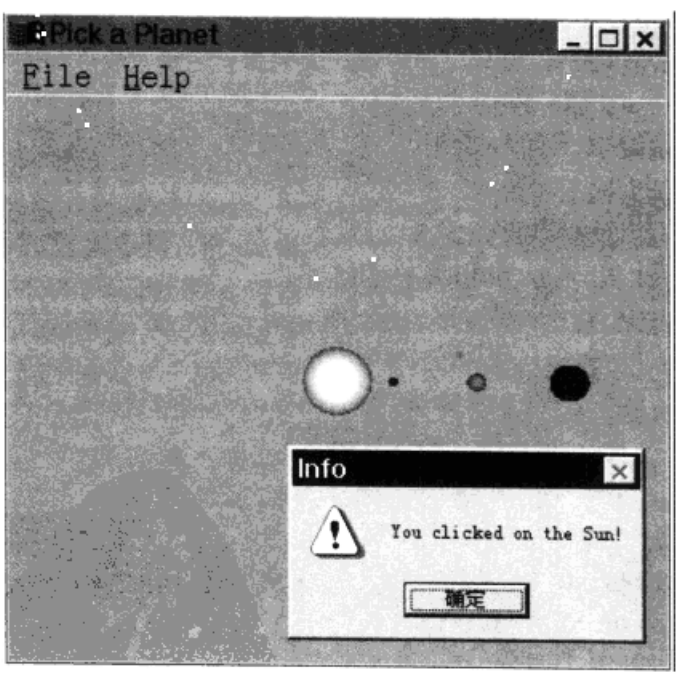


图 14-1　程序 14-1 的运行结果

glInitNames 函数初始化并清除名称堆栈。glPushName 把 0 推入堆栈，使堆栈中至少有一个条目。对于太阳和行星，调用 glLoadNames 命令要绘制的物体，用这一名称替换堆栈最顶层当前名称。本章后面的内容将介绍如何保存堆栈中的名称。

14.1.2 使用选择模式

前面提到过，OpenGL 有三种不同绘图模式，缺省模式是 GL_RENDER，即在屏幕上绘图。只有在调用下面的函数后，OpenGL 才处于选择模式。

```
glRenderModel(GL_SELECTION);
```

当需要在屏幕上绘图时，还要调用

```
glRenderModel(GL_RENDER);
```

使 OpenGL 回到绘图模式。OpenGL 操作的第三种模式为反馈(将在本章后面的内容做介绍)。

只有在事先进入了选择模式时，程序 14-1 的命名程序才能起作用。在多数情况下，用于绘图模式(GL_RENDER)的同一程序，也可在选择模式(GL_SELECTION)下绘制场景。

下面的程序段从 Param 读入点击鼠标器左键的位置并传递给 ProcessSelect 的函数。

```
case WM_LBUTTONDOWN:
    {
    int xPos = LOWORD(lParam);  // horizontal position of cursor
    int yPos = HIWORD(lParam);  // vertical position of cursor

    // Render in selection mode and display results
    ProcessSelection(xPos, yPos);
    }
```

14.1.3 选择缓冲区

选择缓冲区存贮着绘图过程中的命中记录。只要所绘制的一个几何要素或一组几何要素包括在取景体积中，就生成命中记录。通常条件下，记录的内容是显示在屏幕上的某一物体或某一部分。

选择缓冲区是一个无符号整数数组。每个命中记录至少占据这个数组的四个单元。数组的第一个单元为发生命中记录时，在名称堆栈上出现名称的数目。对于程序 14-1，为 1，其后的两个单元分别为发生上一次命中记录以来包含在取景体积中所有顶点为 z 坐标的最小值和最大值。存贮到选择缓冲区时，把取值范围从[0,1]放大到$[0,2^{32}-1]$范围内，然后，依次存放其他命中记录。

选择缓冲区的这种格式使用户不必知道要分析多少命中记录，因为只有在调用 glRenderMode 函数时，把绘图模式切换回 GL_RENDER，才写入选择缓冲区。该函数返回拷贝的命中记录。

下面的程序段为点击鼠标器后进行的函数调用。首先给选择缓冲区分配内存，再用 glSelectBuffer 指定内存的大小。这个函数有两个参量，即缓冲区的长度和其指针。

```
// Process the selection, which is triggered by a right mouse
// click at (xPos, yPos).
```

```
#define BUFFER_LENGTH 64
void ProcessSelection(int xPos, int yPos)
    {
    // Space for selection buffer
    GLuint selectBuff[BUFFER_LENGTH];

    // Hit counter and viewport storeage
    GLint hits, viewport[4];

    // Setup selection buffer
    glSelectBuffer(BUFFER_LENGTH, selectBuff);

    // Get the viewport
    glGetIntegerv(GL_VIEWPORT, viewport);

    // Switch to projection and save the matrix
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();

    // Change render mode
    glRenderMode(GL_SELECT);

    // Establish new clipping volume to be unit cube around
    // mouse cursor point (xPos, yPos) and extending two pixels
    // in the vertical and horzontal direction
    glLoadIdentity();
    gluPickMatrix(xPos, yPos, 2,2, viewport);

    // Apply perspective matrix
    gluPerspective(45.0f, fAspect, 1.0, 425.0);

    // Draw the scene
    RenderScene();

    // Collect the hits
    hits = glRenderMode(GL_RENDER);

    // If a single hit occured, display the info.
    if(hits == 1)
        ProcessPlanet(selectBuff[3]);
```

```
// Restore the projection matrix
glMatrixMode(GL_PROJECTION);
glPopMatrix();

// Go back to modelview for normal rendering
glMatrixMode(GL_MODELVIEW);
}
```

14.1.4 拾 取

在选择过程中，用鼠标定位创建并使用一个经过修改的取景体积时，发生拾取操作。在鼠标所在位置创建一个小的取景体积，可以针对处于其中的物体发生命中记录。这样，在选择缓冲区内就可以知道其中是否有物体，鼠标点击在哪一物体上。

gluPickMatrix 函数是一个创建描述新取景体积矩阵有用的应用程序：

```
void gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height,
                                                    GLint viewport[4]);
```

其中(x,y)参数为取景体积中心的窗口坐标。在此可以是鼠标位置，也可以把鼠标位置作为取景体积的中心。width，height 参数为以像素为单位指定窗口中的取景体积。如果在一个物体附近点击鼠标器，可以用一个较大值；而在物体上或紧挨着物体，可用一个较小值。视见区数组中包含当前定义的视见区的窗口坐标，即调用

```
glGetInteger(GL_VIEWPORT, viewport);
```

调用 gluPickMatrix 时，要首先存贮当前投影矩阵状态，然后，调用 glLoadIdentity 函数创建一个单位取景体积。调用 gluPickMatrix 可以把这一取景体积变换到正确位置。最后，一定要进一步应用透视投影，否则，不会得到一个真实映射。

```
// Switch to projection and save the matrix
glMatrixMode(GL_PROJECTION);
glPushMatrix();

// Change render mode
glRenderMode(GL_SELECT);

// Establish new clipping volume to be unit cube around
// mouse cursor point (xPos, yPos) and extending two pixels
// in the vertical and horzontal direction
glLoadIdentity();
gluPickMatrix(xPos, yPos, 2,2, viewport);

// Apply perspective matrix
gluPerspective(45.0f, fAspect, 1.0, 425.0);

// Draw the scene
```

```
    RenderScene();

    // Collect the hits
    hits = glRenderMode(GL_RENDER);
```

在上面的程序段中，首先存贮取景体积，然后，进入选择模式。把取景体积限制在仅包含当前鼠标器指针占据的区域。调用 RenderScene 函数重新绘制场景，然后调用 glRenderMode 函数把 OpenGL 恢复到绘图模式，并返回命中记录数目。

在下面的程序段中，如果出现一个命中记录，则存入选择缓冲区，最后，恢复投影矩阵，并把当前矩阵堆栈切换回模型取景矩阵。

```
    // If a single hit occured, display the info.
    if(hits == 1)
        ProcessPlanet(selectBuff[3]);

    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    // Go back to modelview for normal rendering
    glMatrixMode(GL_MODELVIEW);
```

14.1.5 有层次的拾取

在上面的例子中，并不是把名称推入名称堆栈，而仅仅是替代名称堆栈中已有的名称，因此，返回到选择缓冲区的名称也只是驻留在名称堆栈上的唯一一个名称。在生成命中记录时，如果把多个名称放入名称堆栈，就可以得到多个名称。

为说明返回名称堆栈多个名称的使用方法，对上面的例子做一些修改。图 14-2 所示为两个行星，各有其自己的月亮。现在不仅要标识行星或太阳，还要标识出与月亮相关的行星。在选中月亮时，要把月亮和行星的名称一起推入堆栈。

程序 14-2

```
#define EARTH  1
#define MARS   2
#define MOON1  3
#define MOON2  4

// Called to draw scene

void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
// Save the matrix state and do the rotations
glMatrixMode(GL_MODELVIEW);
glPushMatrix();

// Translate the whole scene out and into view
glTranslatef(0.0f, 0.0f, -300.0f);

// Initialize the names stack
glInitNames();
glPushName(0);

// Draw the Earth
glPushMatrix();
glRGB(0,0,255);
glTranslatef(-100.0f,0.0f,0.0f);
glLoadName(EARTH);
auxSolidSphere(30.0f);

// Draw the Moon
glTranslatef(45.0f, 0.0f, 0.0f);
glRGB(220,220,220);
glPushName(MOON1);
auxSolidSphere(5.0f);
glPopName();
glPopMatrix();

// Draw Mars
glRGB(255,0,0);
glPushMatrix();
glTranslatef(100.0f, 0.0f, 0.0f);
glLoadName(MARS);
auxSolidSphere(20.0f);

// Draw Moon1
glTranslatef(-40.0f, 40.0f, 0.0f);
glRGB(220,220,220);
glPushName(MOON1);
auxSolidSphere(5.0f);
glPopName();

// Draw Moon2
```

```
    glTranslatef(0.0f, -80.0f, 0.0f);
    glPushName(MOON2);
    auxSolidSphere(5.0f);
    glPopName();
    glPopMatrix();

    // Restore the matrix state
    glPopMatrix(); // Modelview matrix

    // Flush drawing commands
    glFlush();
    }
```

下面的程序段用于分析选择缓冲区。因为最先被推入堆栈，行星的名称总在名称堆栈的最底层。若选中了月亮，也要把相应的名称推入堆栈。ProcessPlanet 函数显示选中的行星名称或月亮名称。

```
// Parse the selection buffer to see which planet/moon was selected
void ProcessPlanet(GLuint *pSelectBuff)
    {
    int id,count;
    char cMessage[64];

    // How many names on the name stack
    count = pSelectBuff[0];

    // Bottom of the name stack
    id = pSelectBuff[3];

    // Select on earth or mars, whichever was picked
    switch(id)
        {
        case EARTH:
            strcpy(cMessage,"You clicked Earth.");

            // If there is another name on the name stack,
            // then it must be the moon that was selected
            // This is what was actually clicked on
            if(count == 2)
                strcat(cMessage,"\nSpecifically the moon.");

            break;
```

```
        case MARS:
            strcpy(cMessage,"You clicked Mars.");

            // We know the name stack is only two deep. The precise
            // moon that was selected will be here.
            if(count == 2)
                {
                if(pSelectBuff[4] == MOON1)
                    strcat(cMessage,"\nSpecifically Moon #1.");
                else
                    strcat(cMessage,"\nSpecifically Moon #2.");
                }
            break;

        // If nothing was clicked we shouldn't be here!
        default:
            strcpy(cMessage,"Error - Nothing was clicked on!");
            break;
        }

    // Display the message about planet and moon selection
    MessageBox(NULL,cMessage,"Selection Message",MB_OK);
    }
```

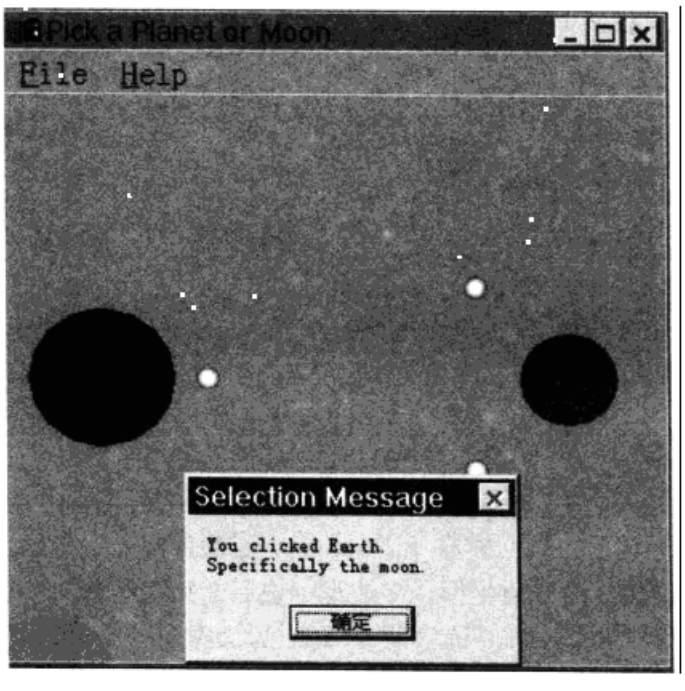


图 14-2　有层次的选择

14.2 反 馈

与选择模式相似，反馈模式也不在屏幕上以像素形式绘制图形，而是把如何创建场景的信息写入反馈缓冲区。这些信息包括在窗口坐标中经过变换的顶点数据、光照计算和纹理化得到的颜色数据。

调用 glRenderMode(GL_FEEDBACK)可以进入反馈模式。与选择模式相似，只有调用 glRenderMode(GL_RENDER)恢复到正常绘图模式下，才能把信息写入反馈缓冲区。

14.2.1 反馈缓冲区

1. 反馈缓冲区

反馈缓冲区是由 glFeedback 函数指定的一个浮点型数组：

```
void glFeedbackBuffer(GLsizei size, GLenum type, GLfloat *buffer);
```

调用上面的函数，可以设定反馈缓冲区的大小，写入缓冲区的信息类型、数据以及指向缓冲区的指针。

表 14-1 给出有效的数据类型。这些数据类型指定在反馈缓冲区写入每一顶点数据的数目。在色彩指数模式下，k 代表一个值，而在 RGBA 模式下，为 4 个值。

表 14-1 glFeedback 函数的有效数据类型

类型参数	坐标	颜色	纹理	总值
GL_2D	x,y	——	——	2
GL_3D	x,y,z	——	——	3
GL_3D_COLOR	x,y,z	k	——	3+k
GL_3D_COLOR_TEXTURE	x,y,z	k	4	7+k
GL_4D_COLOR_TEXTURE	x,y,z,w	k	4	8+k

2. 反馈数据

反馈缓冲区由一系列标记以及紧跟其后的顶点数据、颜色和纹理数据构成。标记表示所绘制的几何要素类型，如表 14-2 所示。

表 14-2 反馈数据

几何要素类型	符号常数	相关数据
点	GL_POINT_TOKEN	顶点
线	GL_LINE_TOKEN 或 GL_LINE_RESET_TOKEN	顶点
多边形	GL_POLYGON_TOKEN	n 个顶点
位图	GL_BITMAP_TOKEN	顶点
像素矩形	GL_DRAW_PIXEL_TOKEN 或 GL_COPY_PIXEL_TOKEN	顶点
用户定义标识	GL_PASS_THROUGH_TOKEN	浮点数

点、位图、像素标记后的数据根据 glFeedback 函数的定义，可以为一个顶点、颜色和纹理数据；线标记后面为 2 组顶点坐标；多边形标记后面为多边形的顶点数目及相应的顶点坐标；用户自定义的标记 GL_PASS_THROUGH_TOKEN 后面跟随一个自定义的浮点值。图 14-3 所示为按 GL_3D 类型设置的反馈缓冲区示意图。

反馈缓冲区	[0]	GL_POINT_TOKEN
	[1]	x 坐标
	[2]	y 坐标
	[3]	z 坐标
	[4]	GL_PASS_THROUGH_TOKEN
	[5]	用户定义值
	[6]	GL_POLYGON_TOKEN
	[7]	顶点数目
	[8]	第一个顶点的 x 坐标
	[9]	第一个顶点的 y 坐标
	[10]	第一个顶点的 z 坐标
	[11]	第二个顶点的 x 坐标
.....		
	[n]	最后一个顶点的 z 坐标

图 14-3　反馈缓冲区示意图

3. GL_PASS_THROUGH_TOKEN

在选择模式下，用户可以给某一几何要素命名进行标记。与此相似，在反馈模式下可以调用 glPassThrough 函数在几何要素间设置标记。

```
void glPassThrough(GLfloat token);
```

该函数把 GL_PASS_THROUGH_TOKEN 放置在反馈缓冲区中，并在其后放置调用函数时用户设定的值。这与在选择模式下命名几何要素相似，是在反馈缓冲区标记几何要素的唯一方式。

14.2.2　一个例子

反馈的一个重要应用是提供有关所绘制物体的窗口坐标信息。根据这些信息，用户可以对窗口中的物体或物体周围的窗口附近放置控件。

本小节举例说明反馈的使用方法。首先用选择机制确定用户选定屏幕上的两个物体，然后，进入反馈模式并重绘场景，得到窗口坐标下的顶点信息。用这些数据，确定物体的 X,Y 值最小值、最大值并在物体周围绘制一个矩形。

1. 标记物体

下面的程序段为程序 14-3 的 RenderScene 函数。在程序中用选择机制选取屏幕上的物体，但与前面所举选择机制例子不同的是，在这个例子中要说明如何用反馈得到足够多关于物体的信息。应该注意的是，调用 glLoadName 函数在选择缓冲区中标记物体后，就要调用 glPassThrough 函数在反馈缓冲区中标记这个物体。

程序 14-3

```
// Called to draw scene
void RenderScene(void)
    {
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
// Save the matrix state and do the rotations
glMatrixMode(GL_MODELVIEW);
glPushMatrix();

// Translate the whole scene out and into view
glTranslatef(-80.0f, 0.0f, -300.0f);

// Initialize the names stack
glInitNames();
glPushName(0);

// Set material color, Yellow
// Cube
glRGB(255, 255, 0);
glLoadName(CUBE);
glPassThrough((GLfloat)CUBE);
auxSolidCube(75.0f);

// Draw Sphere
glRGB(128,0,0);
glTranslatef(130.0f, 0.0f, 0.0f);
glLoadName(SPHERE);
glPassThrough((GLfloat)SPHERE);
auxSolidSphere(50.0f);

// Restore the matrix state
glPopMatrix(); // Modelview matrix

// Flush drawing commands
glFlush();
}
```

2. 对程序 14-3 的说明

步骤 1：选择物体

图 14-4 所示为绘图程序的输出结果：显示一个立方体和一个圆球。用户在其中一个物体上点击鼠标器时，调用 ProcessSelection 函数，如下面的程序段所示。

```
// Process the selection, which is triggered by a right mouse
// click at (xPos, yPos).
#define BUFFER_LENGTH 64
void ProcessSelection(int xPos, int yPos)
    {
```

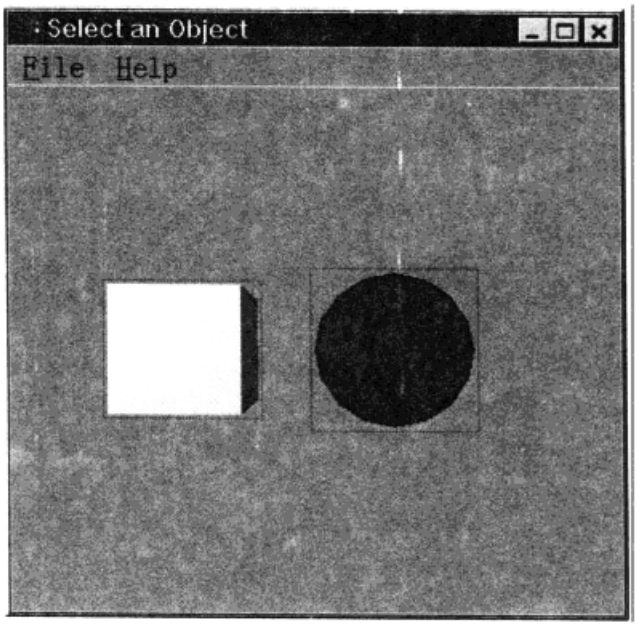


图 14-4 反馈缓冲区的例子

```
// Space for selection buffer
GLuint selectBuff[BUFFER_LENGTH];

// Hit counter and viewport storeage
GLint hits, viewport[4];

// Setup selection buffer
glSelectBuffer(BUFFER_LENGTH, selectBuff);

// Get the viewport
glGetIntegerv(GL_VIEWPORT, viewport);

// Switch to projection and save the matrix
glMatrixMode(GL_PROJECTION);
glPushMatrix();

// Change render mode
glRenderMode(GL_SELECT);

// Establish new clipping volume to be unit cube around
// mouse cursor point (xPos, yPos) and extending two pixels
// in the vertical and horzontal direction
glLoadIdentity();
gluPickMatrix(xPos, yPos, 2,2, viewport);

// Apply perspective matrix
```

```
    gluPerspective(60.0f, fAspect, 1.0, 425.0);

    // Draw the scene
    RenderScene();

    // Collect the hits
    hits = glRenderMode(GL_RENDER);

    // Restore the projection matrix
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    // Go back to modelview for normal rendering
    glMatrixMode(GL_MODELVIEW);

    // If a single hit occured, display the info.
    if(hits == 1)
        MakeSelection(selectBuff[3]);
    }
```

步骤2:

确定了选中的物体后，设置反馈缓冲区并在反馈模式下重新绘图。下面的程序段设置反馈缓冲区并调用 RenderScene 函数重新绘制场景。此时，glPassThrough 函数把该物体的标记写入反馈缓冲区。

```
// Go into feedback mode and draw a rectangle around the object
#define FEED_BUFF_SIZE  4096
void MakeSelection(int nChoice)
    {
    // Space for the feedback buffer
    GLfloat feedBackBuff[FEED_BUFF_SIZE];

    // Storeage for counters, etc.
    int size,i,j,count;

    // Min and max x and y values for 2D vertex positions
    float nMaxX,nMaxY,nMinX,nMinY;

    // Initial minimum and maximum values
    nMaxX = nMaxY = -999999.0f;
    nMinX = nMinY =  999999.0f;

    // Set the feedback buffer
```

```
glFeedbackBuffer(FEED_BUFF_SIZE,GL_2D, feedBackBuff);

// Enter feedback mode
glRenderMode(GL_FEEDBACK);

// Redraw the scene
RenderScene();

// Leave feedback mode
size = glRenderMode(GL_RENDER);

// Parse the feedback buffer and get the
// min and max X and Y window coordinates
i = 0;
while(i < FEED_BUFF_SIZE)
     {
     // Search for appropriate token
     if(feedBackBuff[i] == GL_PASS_THROUGH_TOKEN)
          if(feedBackBuff[i+1] == (GLfloat)nChoice)
          {
          i+= 2;
          // Loop until next token is reached
          while(feedBackBuff[i] != GL_PASS_THROUGH_TOKEN)
               {
               // Just get the polygons
               if(feedBackBuff[i] == GL_POLYGON_TOKEN)
                    {
                    // Get all the values for this polygon
                    count = (int)feedBackBuff[++i]; // How many vertices
                    i++;

                    for(j = 0; j < count; j++)    // Loop for each vertex
                         {
                         // Min and Max X
                         if(feedBackBuff[i] > nMaxX)
                              nMaxX = feedBackBuff[i];

                         if(feedBackBuff[i] < nMinX)
                              nMinX = feedBackBuff[i];

                         i++;
```

```
                    // Min and Max Y
                    if(feedBackBuff[i] > nMaxY)
                        nMaxY = feedBackBuff[i];

                    if(feedBackBuff[i] < nMinY)
                        nMinY = feedBackBuff[i];

                    i++;
                    }
                }
            else
                i++; // Get next index and keep looking
            }
        break;
        }
    i++;
    }

// Draw focus rectangle
HighLight((int)floor(nMinX+0.5), (int)floor(nMinY+0.5),
    (int)floor(nMaxX+0.5), (int)floor(nMaxY+0.5));
}
```

信息写入反馈缓冲区后，查找 GL_PASS_THROUGH_TOKEN 标记，判断下一个值是否是所查找的。然后，读该物体所有多边形的信息，得到 X,Y 值的最小、最大值。HighLight 函数使用 Win32 的 DrawFocusRect 函数在所选中的物体周围画一个矩形。在 XOR 模式下绘图，调用该函数两次，矩形框会消失。这样，可以在物体上点击一次，选中该物体，再点击一次，不选中这一物体。

14.3 相关函数

```
void glInitNames(void);
void glLoadName(GLuint name);
void glPushName(GLuint name);
void glPopName(void);
void glPassThrough (GLfloat token);
GLint glRenderMode(GLenum mode);
void glSelectBuffer(GLsizei size, GLuint *buffer);
void gluPickMatrix (GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4]);
void glFeedbackBuffer (GLsizei size, GLenum type, GLfloat *buffer);
```

第十五章　OpenGL 编程技巧

前面的所有章节都是针对 OpenGL 的某一个方面内容做详细的介绍，侧重点在于讲述 OpenGL 的函数使用方面的问题，使初学者能够对 OpenGL 有一个全面的认识。在实际应用中，使用 OpenGL 是在具体的程序中得到体现的。本章的目的是讲述在目前流行的编程语言中，如何实现 OpenGL 的功能，使读者能够比较顺利地实现 OpenGL 的功能，得到满足需要的三维场景。

15.1　基于 MFC 的 OpenGL 编程

本节的目的在于为使用流行应用程序框架的 C++程序员提供一个 OpenGL 编程的起点。本节使用的应用程序框架是 MFC(Microsoft Foundation Classes)，本节的例子和屏幕显示结果都是由 Microsoft Visual C++ 5.0 得到的。

为顺序完成本节所讲述的内容，应该熟悉下列内容:

(1) 用于 Windows NT 和 Windows 95 编制应用程序的 Visual C++和 MFC。

(2) 本书第四章中有关 Windows 环境下的 OpenGL 和绘图描述表使用的内容。

(3) 本书第七章中有关使用调色板的内容。

15.1.1　从应用程序中分离 OpenGL 程序

设计应用程序时，应该尽量使源程序模块化。这样，便于调制程序，也提高各功能模板的再利用效率。把 OpenGL 源程序写入一个单独的模块可以在保持应用程序其他功能的情况下，仅改写这部分源程序。本节把简单的 C 程序用 MFC 和应用程序转换成 C++程序。

首先，声明 glcode.c 源文件中的三个函数。glcode.h 包含这些函数的声明并由 CView 导出类文件调用。

```
// glcode.h
// Declarations for external OpenGL module.  These functions are
// defined in glcode.c and are called appropriately by the CView
// derived classes.


extern "C" {

    void GLSetupRC(void *pData);
    void GLRenderScene(void *pData);
    void GLResize(GLsizei h, GLsizei w);
```

其中 GLSetupRc 函数用于绘图描述表初始化，既可以简单到仅有设定清除用颜色的功能，

也可以复杂到能够创建光照条件的程度； GLRenderScene 函数由 CView 导出类的 OnDraw 子函数调用，完成绘图操作过程；GLResize 函数由 WM_SIZE 处理程序调用，传递窗口绘图区的宽、高值并对已创建的取景体积和视见区做必要的计算。

值得注意的是，GLSetupRc 和 GLRenderScene 函数为空指针，可以在不改变接口的情况下，把任意数据类型数据传递给绘图程序。尽管可以使 glcode.c 文件成为一个 C++文件，但是把一个已有的 C 源程序移入 MFC 程序实现起来更容易。Visual C++把这个模块按 C 文件编译并与应用程序的其余部分联接起来。

15.1.2 启动 Appwizard

许多用 Visual C++编写的应用程序都要从 AppWizard 开始，AppWizard 可以在很短时间内给出一个功能齐全的单一文档界面、多文档界面，或有对话框的应用程序界面，因此，有必要介绍如何创建一个使用 OpenGL 的单一文档界面 MFC 应用程序。创建一个 OpenGL 场景，需要向 CView 类中加入特征和功能。同样的方法也适用于向 CWnd 导出类加入 OpenGL 功能。

1. 创建界面

下面用 AppWizard 创建一个单一文档界面，并忽略有关数据库和 OLE 选项，如图 15-1 所示。同时，因为 OpenGL 仅仅能直接在位平面大于 4 的彩色打印机设备描述表上绘制场景，有必要关闭打印和打印预览选项。

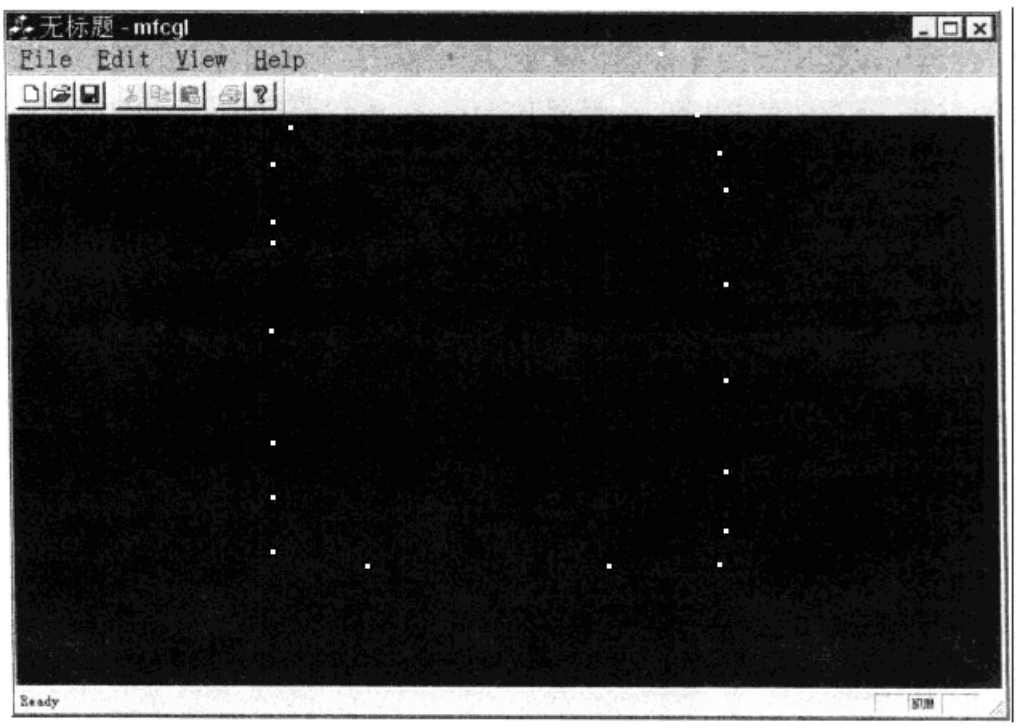


图 15-1 单一文档界面

2. 添加库

向上述界面添加 OpenGL 源程序前，要在项目中添加 OpenGL 库。这可以选择主菜单的 Project/Settings 对话框来实现，如图 15-2 所示。应用程序类型不同，就要添加不同的库。图中所示仅为使用 OpenGL 函数时必需的两个库：opengl32.lib，glu32.lib。

同时，要在项目中添加 OpenGL 头文件。

```
#include <gl\gl.h>
#include <gl\glu.h>
```

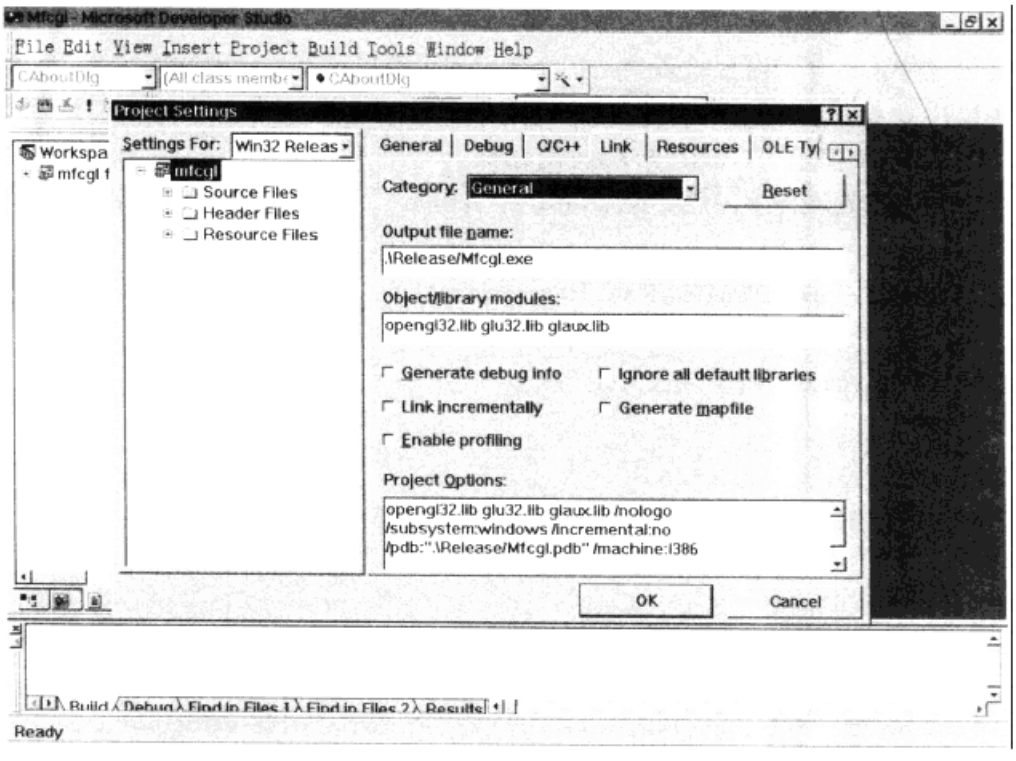


图 15-2　添加 OpenGL 库

15.1.3　MFC 程序设计的注意事项

1. 准备好 CView

首先要把使用 OpenGL 的窗口风格设置成 WS_CLIPCHILDREN 和 WS_CLIPSIBINGS。

```
BOOL CMfcglView::PreCreateWindow(CREATESTRUCT& cs)
{
    // Add Window styles required for OpenGL before window is created
    cs.style |= (WS_CLIPCHILDREN | WS_CLIPSIBLINGS | CS_OWNDC);

    return CView::PreCreateWindow(cs);
}
```

PreCreateWindow 在 mfcglView.cpp 文件中，是 CView 导出类的虚拟子函数。这个函数使用户可以在创建窗口前修改 CREATESTRUCT 信息。这个结构的一个成分贮存所创建窗口的风格。窗口的风格由字位的逻辑或设定。其中字位 CS_OWNDC 使窗口有自己的设备描述表。有些 MFC 函数返回的设备描述表指针只是临时性的，因此，应该把它们保存起来。这样，可以节省时间、提高效率。

CMfglView 类所分配的空间用于存贮设备描述表和绘图描述表，设备描述表和绘图描述表的声明如下：

```
public:
```

```
    HGLRC m_hRC;
    HDC m_hDC;
```

2. 像素格式和绘图描述表

因为创建像素格式时需要设备描述表，在设置了窗口风格后，就应该设置 OpenGL 像素格式。可以用 Class Wizard 添加窗口接受 WM_CREATE 信息时调用信息映射。图 15-3 所示为还包含 WM_DESTORY 信息的 Class Wizard 对话框。

在 WM_CREATE 处理程序内设置像素格式相对简单。下面的程序段为设备描述表选择像素格式程序。

```
int CMfcglView::OnCreate(LPCREATESTRUCT lpCreateStruct)
    {
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    int nPixelFormat;                       // Pixel format index
    m_hDC = ::GetDC(m_hWnd);              // Get the Device context

    static PIXELFORMATDESCRIPTOR pfd = {
        sizeof(PIXELFORMATDESCRIPTOR),    // Size of this structure
        1,                                    // Version of this structure
        PFD_DRAW_TO_WINDOW |              // Draw to Window (not to bitmap)
        PFD_SUPPORT_OPENGL |              // Support OpenGL calls in window
        PFD_DOUBLEBUFFER,                 // Double buffered mode
        PFD_TYPE_RGBA,                    // RGBA Color mode
        24,                                   // Want 24bit color
        0,0,0,0,0,0,                      // Not used to select mode
        0,0,                              // Not used to select mode
        0,0,0,0,0,                            // Not used to select mode
        32,                                   // Size of depth buffer
        0,                                    // Not used to select mode
        0,                                    // Not used to select mode
        PFD_MAIN_PLANE,                       // Draw in main plane
        0,                                    // Not used to select mode
        0,0,0 };                          // Not used to select mode

    // Choose a pixel format that best matches that described in pfd
    nPixelFormat = ChoosePixelFormat(m_hDC, &pfd);

    // Set the pixel format for the device context
    VERIFY(SetPixelFormat(m_hDC, nPixelFormat, &pfd));

    // Create the rendering context
```

```
m_hRC = wglCreateContext(m_hDC);

// Make the rendering context current, perform initialization, then
// deselect it
VERIFY(wglMakeCurrent(m_hDC,m_hRC));
GLSetupRC(m_hDC);
wglMakeCurrent(NULL,NULL);

// Create the palette if needed
InitializePalette();

SetTimer(101,175,NULL); // 8 times a second...

return 0;
}
```

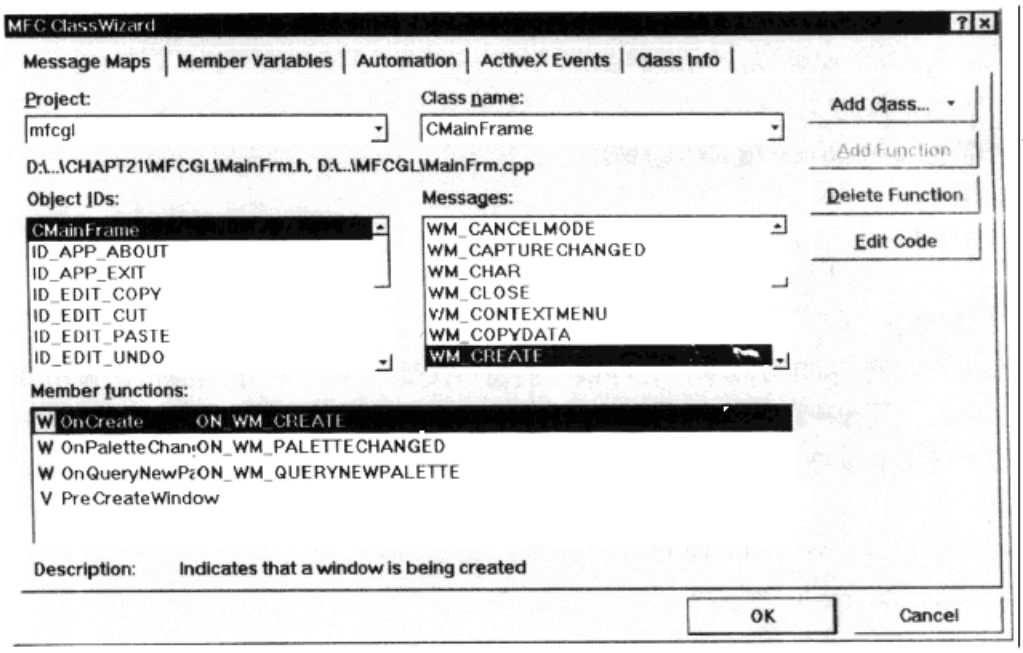


图 15-3 Class Wizard 对话框

值得注意的是，类变量 m_hDC 和 m_hRC 存贮设备描述表和绘图描述表。在创建绘图描述表后，要把它设为当前值并调用 GLSetupRC 外部函数。该函数用于初始化绘图描述表，然后，把它变成非当前值。这样，可以允许多个 OpenGL 的窗口使用多个绘图描述表。

现在在图 15-3 所示的 WM_DESTORY 处理程序中加入清除和删除绘图描述表的程序，并释放为该窗口提供的设备描述表。

```
// The window is being destroyed, delete the rendering context,
// and release the device context
void CMfcglView::OnDestroy()
    {
```

```
    // Kill the timer
    KillTimer(101);

    // Clean up rendering context stuff
    wglDeleteContext(m_hRC);
    ::ReleaseDC(m_hWnd,m_hDC);

    CView::OnDestroy();
    }
```

3. 处理窗口尺寸变化

当窗口尺寸发生变化时，WM_SIZE 信息传入该窗口。这时，可以使用 Class Wizard 为该信息加入一个处理程序，并调用外部函数 GLResize 把新的宽、高值传给该窗口。在调用该函数时，必须把绘图描述表设为当前值，否则，GLResize 函数中的 OpenGL 函数调用不会影响该绘图描述表。

```
void CMfcglView::OnSize(UINT nType, int cx, int cy)
    {
    CView::OnSize(nType, cx, cy);

    VERIFY(wglMakeCurrent(m_hDC,m_hRC));
    GLResize(cx, cy);
    VERIFY(wglMakeCurrent(NULL,NULL));
    }
```

4. 绘制场景

现在加入绘制 OpenGL 场景的程序段。当窗口接受了 WM_PAINT 信息时，调用 OnDraw 子函数。下面的程序段把绘图描述表变成当前值并调用仅有 OpenGL 函数的 GLRenderScene 函数。因为要使用双缓冲窗口，调用 SwapBuffers 函数并把该绘图描述表变成 NULL，即不再为当前值。

```
// Called when window receives WM_PAINT, render our scene
void CMfcglView::OnDraw(CDC* pDC)
    {
    // Make the rendering context current
    wglMakeCurrent(m_hDC,m_hRC);

    // Call our external OpenGL code
    GLRenderScene(NULL);

    // Swap our scene to the front
    SwapBuffers(m_hDC);

    // Allow other rendering contexts to co-exist
    wglMakeCurrent(m_hDC,NULL);
```

```
}
```

在窗口尺寸发生变化或无效时，MFC 在重新绘制场景之后，要清除窗口背景。因为程序中背景为黑色，而清除的屏幕为红色，所以，每次调用 OnDraw 函数屏幕都会闪烁。为消除这一现象，就要改变 WM_ERASEBACKGROUND 的缺省处理程序。可以通过把 CView::OnEraseBkgnd(pDC)的返回值设为 FALSE 来实现。

```
// Override to keep the background from being erased everytime
// the window is repainted
BOOL CMfcglView::OnEraseBkgnd(CDC* pDC)
    {
    return FALSE;
    }
```

5. 使用调色板

要完成的最后一项工作是在 256 色模式下创建并实现 RGB 调色板。与第七章中的做法不同的是，需要创建一个 CPalette 类型的 MFC 对象。

下面的程序为创建和初始化 CPalette 的程序段，向 CMfcGLView 添加的子函数 InitializePalett 与第七章的 GetOpenGLPalette 相似，唯一的差别在于 CPalette 对象是 MFC 构造的，而不是返回的调色板指针。

```
// Initializes the CPalette object
void CMfcglView::InitializePalette(void)
    {
    PIXELFORMATDESCRIPTOR pfd;    // Pixel Format Descriptor
    LOGPALETTE *pPal;             // Pointer to memory for logical palette
    int nPixelFormat;             // Pixel format index
    int nColors;                  // Number of entries in palette
    int i;                            // Counting variable
    BYTE RedRange,GreenRange,BlueRange;    // Range for each color entry (7,7,and 3)

    // Get the pixel format index and retrieve the pixel format description
    nPixelFormat = GetPixelFormat(m_hDC);
    DescribePixelFormat(m_hDC, nPixelFormat, sizeof(PIXELFORMATDESCRIPTOR), &pfd);

    // Does this pixel format require a palette?  If not, do not create a
    // palette and just return NULL
    if(!(pfd.dwFlags & PFD_NEED_PALETTE))
        return;

    // Number of entries in palette.  8 bits yeilds 256 entries
    nColors = 1 << pfd.cColorBits;

    // Allocate space for a logical palette structure plus all the palette entries
    pPal = (LOGPALETTE*)malloc(sizeof(LOGPALETTE) +nColors*sizeof(PALETTEENTRY));
```

```
// Fill in palette header
pPal->palVersion = 0x300;          // Windows 3.0
pPal->palNumEntries = nColors; // table size

// Build mask of all 1's.  This creates a number represented by having
// the low order x bits set, where x = pfd.cRedBits, pfd.cGreenBits, and
// pfd.cBlueBits.
RedRange = (1 << pfd.cRedBits) -1;
GreenRange = (1 << pfd.cGreenBits) - 1;
BlueRange = (1 << pfd.cBlueBits) -1;

// Loop through all the palette entries
for(i = 0; i < nColors; i++)
     {
     // Fill in the 8-bit equivalents for each component
     pPal->palPalEntry[i].peRed = (i >> pfd.cRedShift) & RedRange;
     pPal->palPalEntry[i].peRed = (unsigned char)(
          (double) pPal->palPalEntry[i].peRed * 255.0 / RedRange);

     pPal->palPalEntry[i].peGreen = (i >> pfd.cGreenShift) & GreenRange;
     pPal->palPalEntry[i].peGreen = (unsigned char)(
          (double)pPal->palPalEntry[i].peGreen * 255.0 / GreenRange);

     pPal->palPalEntry[i].peBlue = (i >> pfd.cBlueShift) & BlueRange;
     pPal->palPalEntry[i].peBlue = (unsigned char)(
          (double)pPal->palPalEntry[i].peBlue * 255.0 / BlueRange);

     pPal->palPalEntry[i].peFlags = (unsigned char) NULL;
     }

// Create the palette
m_GLPalette.CreatePalette(pPal);

// Go ahead and select and realize the palette for this device context
SelectPalette(m_hDC, (HPALETTE)m_GLPalette, FALSE);
RealizePalette(m_hDC);

// Free the memory used for the logical palette structure
free(pPal);
}
```

下面的程序段使用 Class Wizard 添加 WM_QUERYNEWPALETTE 和 WM_PALETTECHANGED 信息的响应函数。

```
BOOL CMfcglView::OnQueryNewPalette()
    {
    // If the palette was created.
    if((HPALETTE)m_GLPalette)
        {
        int nRet;

        // Selects the palette into the current device context
        SelectPalette(m_hDC, (HPALETTE)m_GLPalette, FALSE);

        // Map entries from the currently selected palette to
        // the system palette.  The return value is the number
        // of palette entries modified.
        nRet = RealizePalette(m_hDC);

        // Repaint, forces remap of palette in current window
        InvalidateRect(NULL,FALSE);

        return nRet;
        }
    return CView::OnQueryNewPalette();
    }

void CMfcglView::OnPaletteChanged(CWnd* pFocusWnd)
    {
    if(((HPALETTE)m_GLPalette != NULL) && (pFocusWnd != this))
        {
        // Select the palette into the device context
        SelectPalette(m_hDC,(HPALETTE)m_GLPalette,FALSE);

        // Map entries to system palette
        RealizePalette(m_hDC);

        // Remap the current colors to the newly realized palette
        UpdateColors(m_hDC);
        return;
        }

    CView::OnPaletteChanged(pFocusWnd);
    }
```

在此实现调色板的程序与第七章非常相似，但 Windows 并没有把这些信息发送到 CView 导出类，而是给了应用程序的 CMainFrame 类。因为 Windows 仅向应用程序主窗口发送调色板信息，所以由这个窗口的响应程序把这些信息传给指定的子窗口。

下面的程序使用 Class Wizard 向 CMainFrame 类添加两个调色板信息处理程序。这些处理程序查找激活的视图并向其传递调色板信息。

```
// Route message to CView derived class
void CMainFrame::OnPaletteChanged(CWnd* pFocusWnd)
    {
    CView* pView = GetActiveView();
    if (pView)
        {
        // OnPaletteChanged is not public, so send a message.
        pView->SendMessage(WM_PALETTECHANGED,
                           (WPARAM)(pFocusWnd->GetSafeHwnd()),
                           (LPARAM)0);
        }
    }

// Route message to CView derived class
BOOL CMainFrame::OnQueryNewPalette()
    {
    CView* pView = GetActiveView();
    if (pView)
        {
        // OnQueryNewPalette is not public, so send a message.
        return pView->SendMessage(WM_QUERYNEWPALETTE,
                                  (WPARAM)0,
                                  (LPARAM)0);
        }
    return FALSE;
    }
```

15.2 OpenGL 的 Visual Basic 编程

前面讲述了如何在C程序中实现OpenGL API 调用。本节将介绍在Microsoft Visual Basic 5.0 环境下，如何使用 OpenGL OCX (OLE custom control)。在学习本节内容之前，应该熟悉 Visual Basic 环境以及使用、调用 OCX 的方法。即使读者没有使用 OCX 控件的经验，也会很快发现它是非常易于使用的。

15.2.1 使用 OCX

只要支持 Win32 API 和包括在 DLL 文件中库函数的低层访问，在任何一种 Windows 开发语言或环境中都可以使用 OpenGL。OpenGL API 包含在 opengl32.dll 和 glu32.dll 文件中。正如多数 Win32 API 直接访问 user32.dll，gdi32.del 等 DLL 文件一样，用户也可以在高级语言环境中调用 OpenGL 的函数和命令。每种工具和环境访问外部 DLL 的方法都不相同。通常，要定义函数名称及其参量、返回值类型，函数所在的 DLL 文件。

在前面提到的环境中使用 OpenGL 有两个缺点：

(1) 使用上繁琐。针对一个特定环境，需要定义并输出每一个 OpenGL 函数。另外，参量和返回值类型要与所在特定环境的数据类型相匹配，要定义文件中所有状态变量。

(2) Win32 要求使用 OpenGL 的窗口 Windows 风格只能是 WS_CLIPCHILDREN 和 WS_CLIPSIBLINGS。

尽管用 C 语言编写一个 DLL 文件来完成所有 OpenGL 调用并用高级语言调用这些函数是一个最佳解决方案，但对不使用 C/C++语言编程的用户来说，是无法完成的一件难事。

1. 关于 OCX

把程序加入到 OCX 控件，就可以在任一支持 OCX 的环境中使用。常用的基于 MFC 的 C++ 应用程序、Visual Basic 4.0 或更高版本、Borland Delphi 2.0 或更高版本、Power Builder 等都支持 OCX。与 OCX 的接口按该环境语言需求确定，而不必做特殊考虑。

OCX 控件与 OpenGL API 一起使用，可以解决上面提到的在高层语言中调用 OpenGL 的两个缺点。此时，所有函数均已定义并准备好了一个绘图窗口，可以在任何一个支持 OCX 控件的环境中使用 OpenGL。

需要注意的是，并不是所有的环境都支持用于 NURBS 或多边形镶嵌回调的 OpenGL 函数。更有一些环境不对程序编译而仅做解释，也不可能给函数传递指针。只有在能生成与 C 语言调用规则兼容的编译程序的环境下才能使用。

如 Borland Delphi 才能满足上面提出的要求。

2. 使用 OpenGL.OCX

使用 OpenGL.OCX 的方法很简单，只要把这个控件安装到所用环境中，就可以把这个控件放到窗体上，OpenGL 函数和命令一样进行调用。

每一个命令的名称是去掉“gl”前缀的 OpenGL API。这样给 gl 控件命名，可以使 OpenGL 程序与 C 语言程序相似。

这个 OCX 激活来自应用程序的两个事件：

(1) SetupRC，在 OCX 向窗口绘图前首先调用该函数，用于为这个控件设置像素格式和绘图描述表、光照、背景颜色等。

(2) 绘图，每次控件需要绘图时，调用该函数，该函数中包含所有绘图，可以使用 OpenGL 在绘图区内绘图。

3. 使用 OCX 要注意：

(1) 如果应用程序中有多个 OpenGL 控件，OCX 不能总是把每一特定控件的绘图描述表都当做当前值。为此，提供了 MakeCurrent 和 MakeNotCurrent 函数，而把所有 OpenGL 调用(包括 SetupRC 和 Render 函数)放在这两个函数中间。

(2) 在把 OpenGL.OCX 的绘图描述表设为当前值时，可以直接调用 OpenGL API。可以用 Visual C++对 OCX 控件做修改，以加入新版本 OpenGL 中的函数。

(3) 控件激活绘图事件后，即创建并实现了一个 3-3-2 调色板，不必用户自己创建。

(4) 控件绘图的窗口采用双缓冲区技术，所以要看到所绘图像，需调用 SwapBuffers。

4. OpenGL 标识

OpenGL 的函数和命令要使用很多特定的标识和状态变量。通过 OCX 调用 OpenGL API 时，所引用的标识和状态变量基本上与前面各章使用的相同，但是全部用小写字母。OpenGL.OCX 的帮助文件 OpenGL.hlp 给出 gl，glu，glaux 库的所有宏定义。

15.2.2 在 VB5.0 中安装使用 OpenGL.OCX

使用 OCX 之前，需要由操作系统(Windows95，Windows NT)把它注册成一个 OCX，把 OpenGL.OCX 拷贝到系统目录上，并运行 Ocxreg.exe 文件，即

ocxreg.exe OpenGL.ocx. install (或 uninstall)

这些文件可以在与本书配套磁盘中找到。

1. 安装控件

控件经操作系统注册后，需要安装到 Visual Basic 的工具栏上。首先选择主菜单的 Tools 菜单的 Custom Control 选项，选择 WaitGL OLE Control module，如图 15-4 所示，然后按下“OK”按钮。这样，就可以像使用其他控件一样，使用该控件。

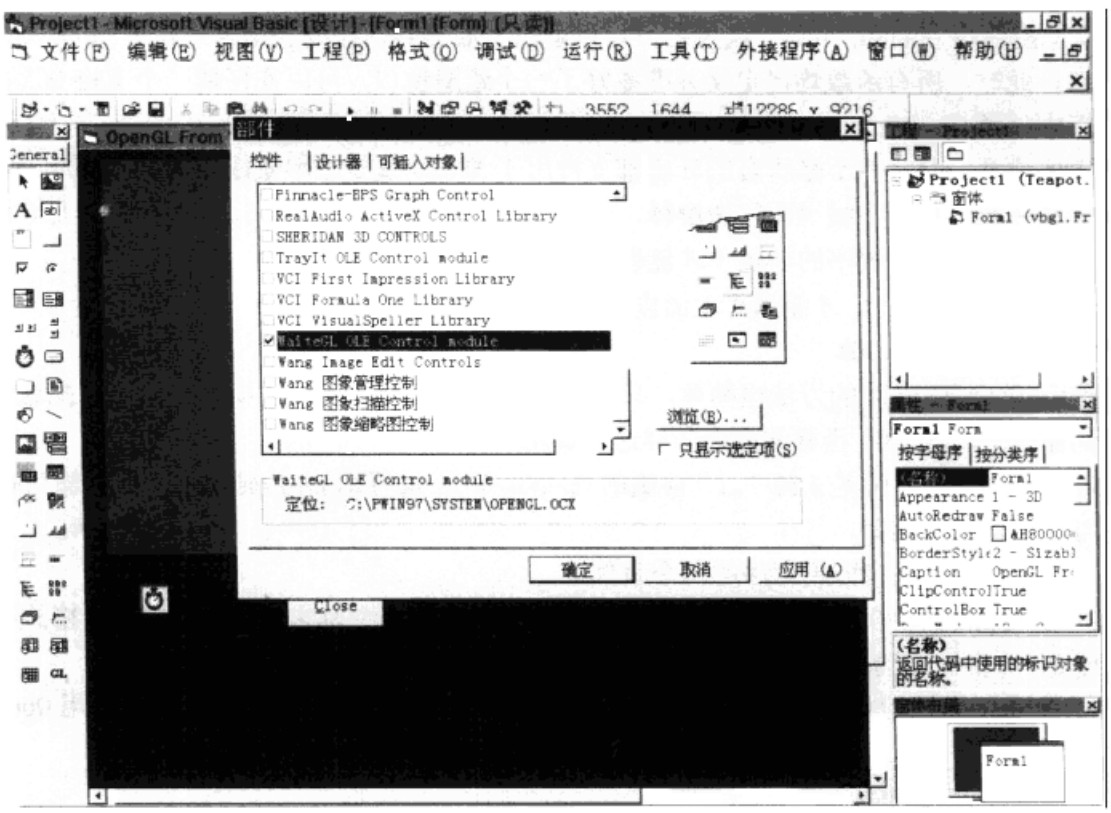


图 15-4 为 Visual Basic 安装控件

2. 一个 Visual Basic 实例

在窗体上，放置一个时间间隔为 200ms 的定时器和 OpenGL 控件，程序运行的结果如图

15-5 所示，但是，由于没有编写绘图程序，此时控件不能在窗口的绘图区上绘图。正如前面讲述的，控件要定义如下两个事件，而这些事件要用户编写程序进行处理。

(1) Setup 事件。下面的程序为设置绘图描述表的程序段，用于设置背景和绘图颜色以及取景体积。

```
Private Sub gl_Render()
    Rem Make the rendering context current
    gl.MakeCurrent

    Rem Clear the screen and draw the aux lib teapot
    gl.Clear (gl.glColorBufferBit)

    gl.auxWireTeapot (55#)

    Rem Flush the commands, make rendering context
    Rem not current, and then finally swap buffers
    gl.Flush
    gl.MakeNotCurrent

    gl.SwapBuffers

End Sub
```

(2) Render 事件。下面的程序为访问 OCX，完成绘图的程序段。在程序中，利用 AUX 库绘制一个茶壶的线框图。

```
Private Sub gl_SetupRC()
    Rem Make the rendering context current
    gl.MakeCurrent

    Rem Set the background color to black
    gl.ClearColor 0#, 0#, 0#, 1#

    Rem Establish the viewing volume
    gl.LoadIdentity
    gl.Ortho -100#, 100#, -100#, 100#, -100#, 100#

    Rem Set the drawing color, flush, and
    Rem make the rendering context not current
    gl.Color 0, 0, 255, 255
    gl.Flush
    gl.MakeNotCurrent

End Sub
```

（3）动画。调用上面的两个函数，就可以在屏幕上显示 OpenGL 图形。本节给出的例子是一个显示动画的程序。定时器的时间间隔为 200ms，每次击发定时器，函数都把显示描述表设为当前值，把取景矩阵选择 5°，然后再把它变为非当前值。

```
Private Sub Timer1_Timer()
    Rem Make the rendeing context current
    gl.MakeCurrent

    Rem Roate 5 degrees
    gl.Rotate 5#, 1#, 1#, 0.5

    Rem Make rendering context not current, then
    Rem force a redraw
    gl.MakeNotCurrent
    gl_Render

End Sub
```

图 15-5　OpenGL 动画

15.3　OpenGL 的 Dephi 编程

1. 安装 OCX 控件

与 Visual Basic 编程相似，为了在 Dephi 环境下调用 OpenGL 函数，也要安装 OCX 控件，即在命令行状态下，键入

```
ocxreg.exe OpenGL.ocx install
```

然后，把这个注册的 OCX 控件安装到 Dephi 的工具板中。具体的步骤如下：

①在主菜单中，选择 Component\install，选择 OCX，如图 15-6 所示；

② 安装 OCX 控件，如图 15-7 所示。

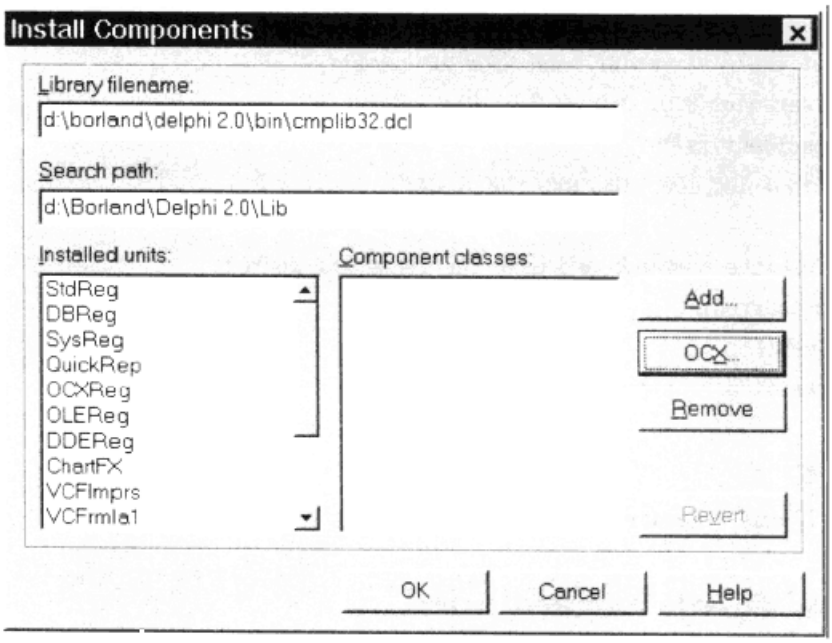


图 15-6　选择菜单

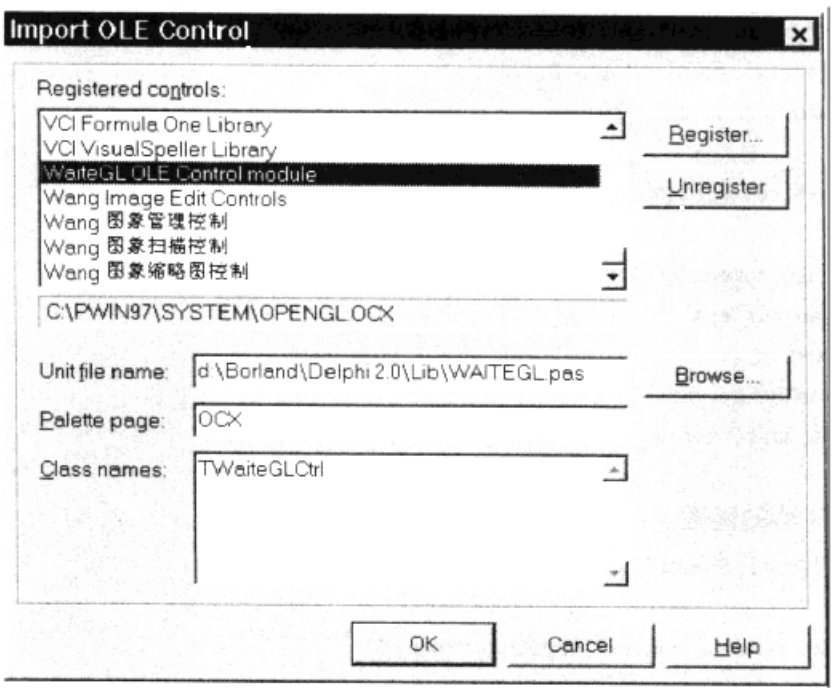


图 15-7　安装 OCX 控件

2．Dephi 的例子

运行本节给出的例子可以得到与上一节 VB 程序相同的结果。作为对比，给出 Dephi 的程序。

①设置绘图描述表的函数

```
procedure TForm1.glSetupRC(Sender: TObject);
```

```
begin
     // Make the Rendering context current
     gl.MakeCurrent();

     // Set the clear color, and viewing volume
     gl.ClearColor(0.0, 0.0, 0.0, 1.0);
     gl.LoadIdentity();
     gl.Ortho(-100,100,-100,100,-100,100);

     // Flush the commands and make the rendering context
     // not current
     gl.Flush();
     gl.MakeNotCurrent();
end;
```

②绘图函数

```
procedure TForm1.glRender(Sender: TObject);
begin
     // Make the rendering context current
     gl.MakeCurrent();

     // Clear the background, and draw a teapot
     gl.Clear(gl.glColorBufferBit());
     gl.Color(0, 0, 255, 255);

     gl.auxWireTeapot(55.0);

     // Flush commands, free rendering context, and
     // swap buffers
     gl.Flush();
     gl.MakeNotCurrent();
     gl.SwapBuffers();
end;
```

③设置定时器的函数

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
     // Make rending context current, then
     // rotate the scene somewhat
     gl.MakeCurrent();
     gl.Rotate(5.0,0.0,1.0,0.5);
     gl.MakeNotCurrent();

     // Repaint the OCX
     gl.Invalidate();
end;
```

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTA0NjUxMTAuemlw",
  "filename_decoded": "10465110.zip",
  "filesize": 16705111,
  "md5": "1b05d17ed391b7c56fa93badd94fde52",
  "header_md5": "cfdef4fc1122002bd03a1111c2fc8b98",
  "sha1": "b38f756a0adbb57c0bba2a2625d38bb211c1581b",
  "sha256": "5b870958bb908a0f66b7f8cb06b60b2f4669cacc35b34f6b58781110061314dd",
  "crc32": 3432402060,
  "zip_password": "",
  "uncompressed_size": 17690509,
  "pdg_dir_name": "",
  "pdg_main_pages_found": 264,
  "pdg_main_pages_max": 264,
  "total_pages": 272,
  "total_pixels": 382732600,
  "pdf_generation_missing_pages": false
}